Math

8

UIUCDCS-R-77-865

UILU-ENG 77 1728

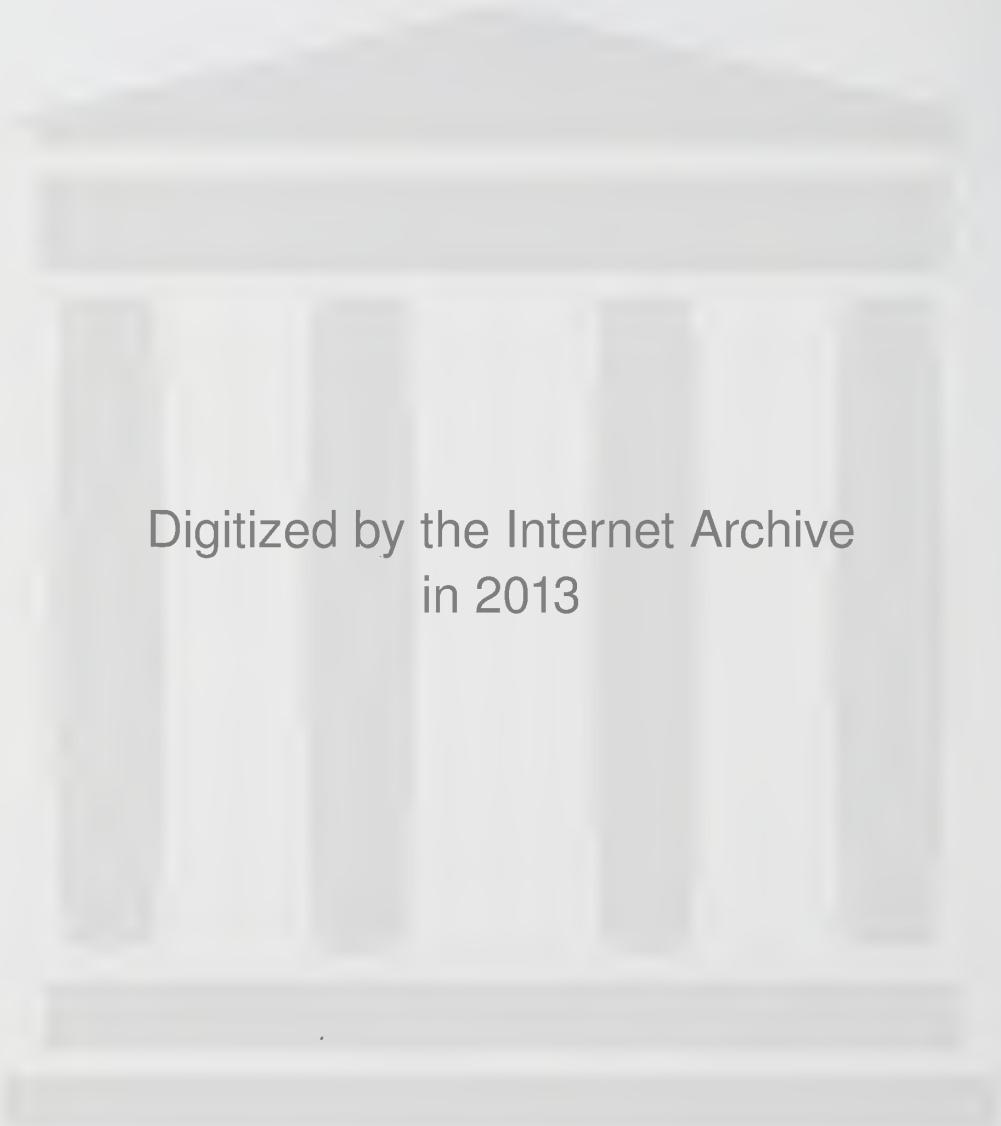
A GRAPHICALLY-PROGRAMMED, MICROPROCESSOR-BASED INDUSTRIAL CONTROLLER

by

Alfred Charles Weaver

May, 1977

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS



A GRAPHICALLY-PROGRAMMED, MICROPROCESSOR-BASED INDUSTRIAL CONTROLLER

BY

ALFRED CHARLES WEAVER

B.S., University of Tennessee, 1971
M.S., University of Illinois, 1973

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1976

Urbana, Illinois

A GRAPHICALLY-PROGRAMMED, MICROPROCESSOR-BASED INDUSTRIAL CONTROLLER

Alfred Charles Weaver, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1976

While programmable industrial controllers have benefited from continuous technological improvements since their introduction in 1969, considerably less effort has been expended on the equally important questions of how to program an industrial controller, what type of programming language to use, and how the controller should interface with its human user. The advent of the microprocessor has encouraged speculation that its use in an industrial controller can both simplify system hardware and expand system flexibility by permitting sophisticated system software.

This thesis discusses the applicability of microprocessors to an industrial environment, traces the development of process control software, and suggests a new, graphic programming language based on familiar relay symbols as the system's input. A design is presented for two stand-alone, microprocessor-based machines--the controller itself, which interprets a user program and manages system I/O, and an auxiliary program loader which supervises interactive graphic programming using a special keyboard and CRT. When connected, the two units communicate to provide the user with the capability of monitoring and changing a running system.

ACKNOWLEDGMENTS

Special thanks are due to the chairman of my doctoral committee, Professor T. A. Murrell, for his continuous encouragement and guidance throughout this research. His contributions to the system software, including the original definition of the column-oriented internal code described herein and simplifications to the timer/counter module, are appreciated and gratefully acknowledged.

Also, many thanks to Donald Henry and Marvin Schilt of Struthers-Dunn, Inc., Bettendorf, Iowa, whose experience and expertise in the area of process control provided constant guidance as we made critical decisions regarding the nature of this system. The generous encouragement and support of Struthers-Dunn, Inc., who provided some software packages and computer time for simulation, are greatly appreciated.

I appreciate the combined efforts of Cinda Robbins, Shiela Morgan, and Cathy Gallion who typed this report, and Stanley Zundo who produced all the drawings. Special thanks are also in order for Mary Jane Irwin, who reviewed and corrected the entire manuscript numerous times.

TABLE OF CONTENTS

List of Figures

List of Figures (Continued)

1. INDUSTRIAL PROCESS CONTROLLERS

1.1 Introduction to Process Controllers

The process control systems of the early 1960s were essentially all-relay systems and suffered from a number of critical deficiencies:

1. Each new application meant a new, custom design;
2. Each new design implied expensive field wiring;
3. Hardware costs were high;
4. Relay hardware was unreliable without extensive field maintenance; and
5. Systems, once installed, offered no flexibility for changing or upgrading the control mechanism.

The development of the microprocessor, coupled with the adaptation of computer science software technology, is bringing forth a new generation of industrial controllers which substitutes programming for field wiring and CRT debug monitors for continuity testers.

Programmable controllers are useful in a host of industrial applications. Some typical applications include:

conveying systems	sorting mechanisms
packaging operations	pumping stations
warehousing	assembly operations
palletizing	production testing
gaging and testing	material handling
labeling	filter cycling
transfer lines	batch sequencing

machine tools	food processing
welding controls	baggage routing

It should be noted at the outset that industrial controllers, while highly versatile, are not equipped to solve all types of industrial control problems. The continuous control of a chemical plant, for instance, is highly dependent upon continuous feedback, sampling rates, and solutions to systems of differential equations. The industrial control system described herein, named the Director 1001 and hereafter called the D-1001, is adequate for all the "typical" applications listed above, but in general is limited to the control of those machines and processes which can be mathematically modeled as a sequential machine, which are dependent upon 20-100 independent variables, and which do not require extensive feedback in the continuous control sense.

While the major thrust of this thesis is the design of the system software for such a controller (determination of an effective programming language and design of two operating systems, a graphic translator, and a communications package between dual processors), a successful implementation requires that due consideration be given to the hardware environment in which the software is to operate. By designing both hardware and software simultaneously, we have attempted to achieve the proper balance between functions implemented in hardware and those implemented in software.

1.2 Function of an Industrial Controller

Reduced to its most primitive function, a controller is expected to perform the equivalent of the iterative solution of a set of Boolean

equations. The Boolean equations relate variables of three types:

1. INPUT variables, whose values are supplied by the real world and can never be changed directly by the controller. Typical inputs are the status of limit switches, pushbuttons, pressure switches, and motion detectors.
2. CONTROL variables, whose values are calculated as functions of inputs, outputs, and other control variables, but whose values are not accessible from the real world. With regard to programming, control variables are often an intermediate result. They may be used for optimization (common subexpression elimination), to create easily monitored test points in critical control sequences, or simply to reduce the complexity of very involved logical operations.
3. OUTPUT variables, whose values are calculated as functions of input, control, and other output variables. Typical outputs are the status of lights, solenoids, motor starters, safety brakes, and annunciators.

Whether the controller actually constructs or solves a sequence of Boolean equations is of no importance as long as an equivalent solution is found and one can unambiguously specify an equivalent Boolean expression for the control function implemented.

Of critical importance to the validity of the computed results is the rate at which the iterative solution is repeated. While the required frequency of solution varies from application to application,

a generally acceptable range is 10 to 100 iterations per second [1, 2].

In addition to performing its minimal requirement of providing a continuous set of solutions to sequences of Boolean equations, a controller should provide some capability for digital operations such as timing and counting. The ability to manipulate some digital information, in addition to strictly Boolean variables, is requisite for all but the most trivial control systems.

1.3 Control Sequence Specification

In the earlier industrial controllers the decision-making logic functions were performed with either electro-mechanical relays or custom, hard-wired, static logic devices. The documentation for the control function of such systems was the electrical drawing defining how the relays and/or logic devices were to be interconnected to form the control sequence. This "team" of relays and control diagrams has been used successfully for many years. The control diagram which defines the relationships among the input, output, and control variables is called a "relay ladder diagram" (RLD) due to its characteristic symbols and format.

Described briefly, the RLD uses the leftmost vertical line to represent a source of electrical power ("power line"); current propogates through the matrix as relays enable or inhibit current flow along the horizontal paths. If current does reach an output (indicated by a circle) in the rightmost column the output will turn on, else it will turn off. See Figure 1, a sample RLD, and Figure 2, a legend for some standard symbols.

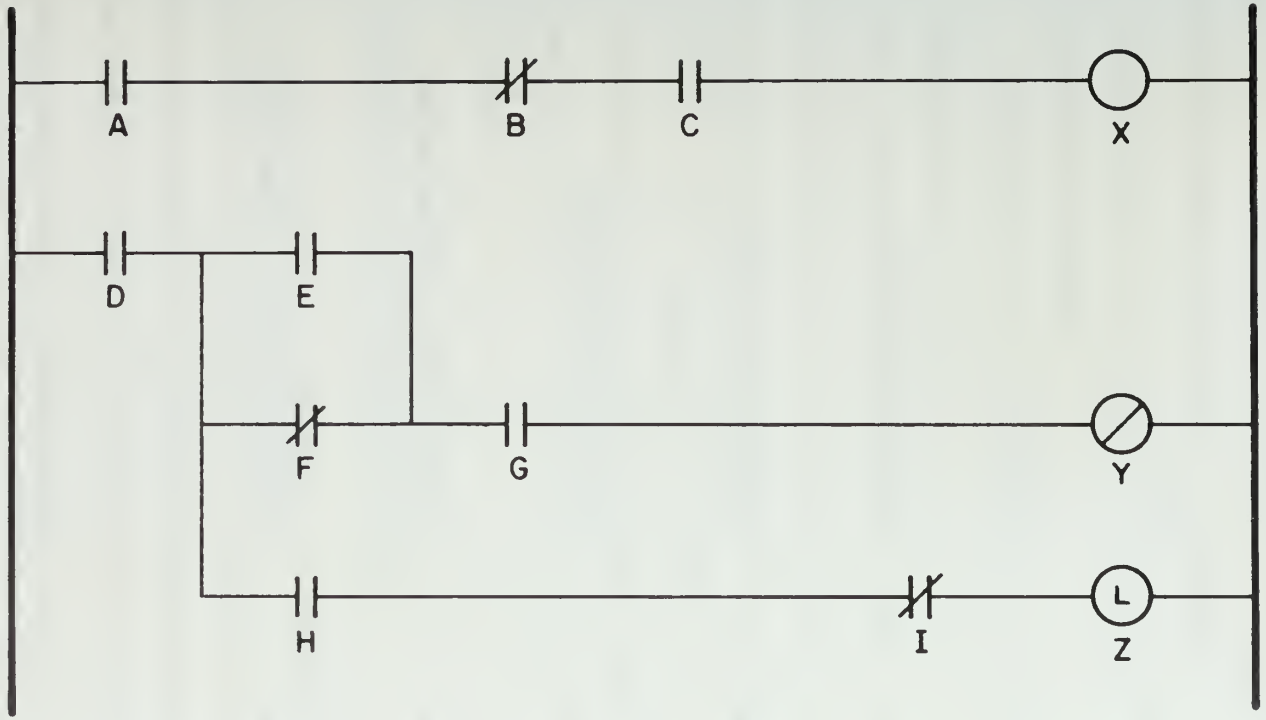


Figure 1. Relay Ladder Diagram

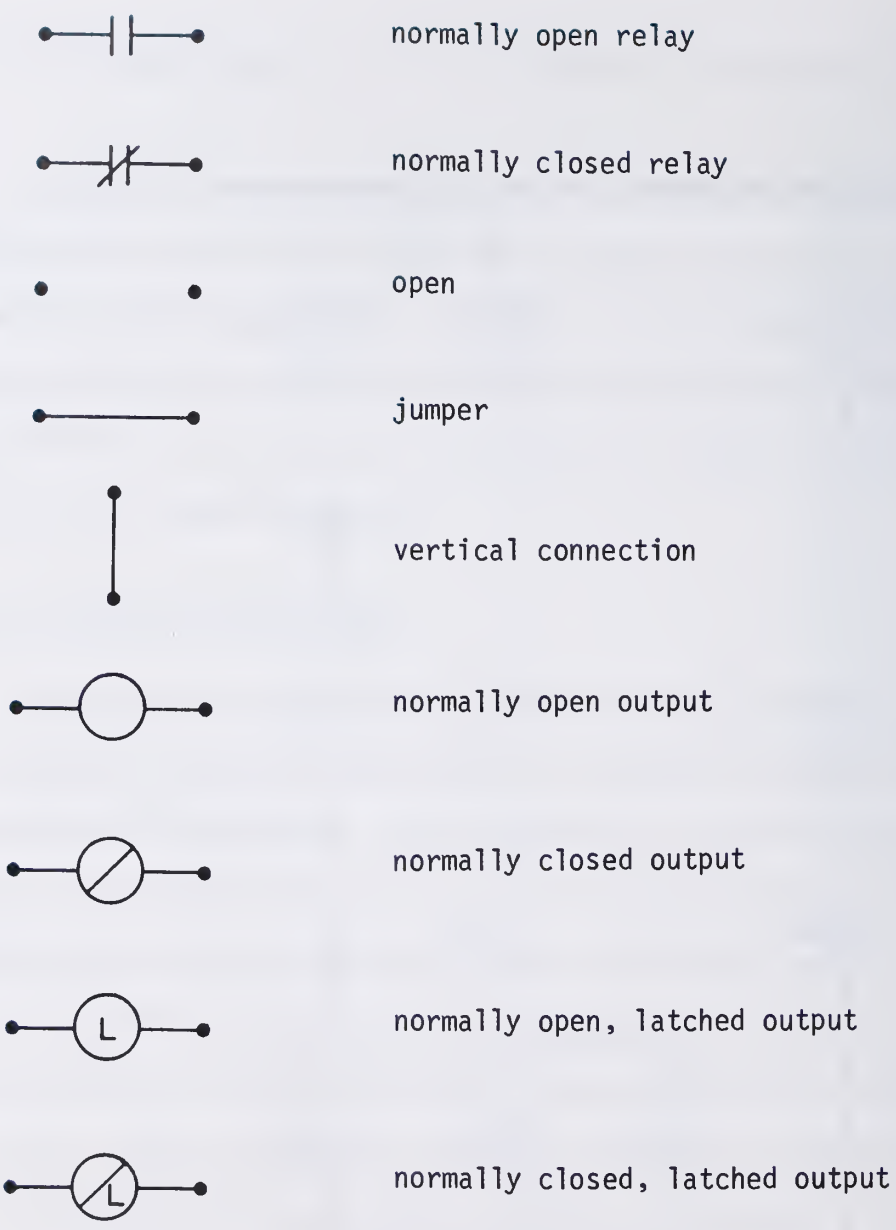


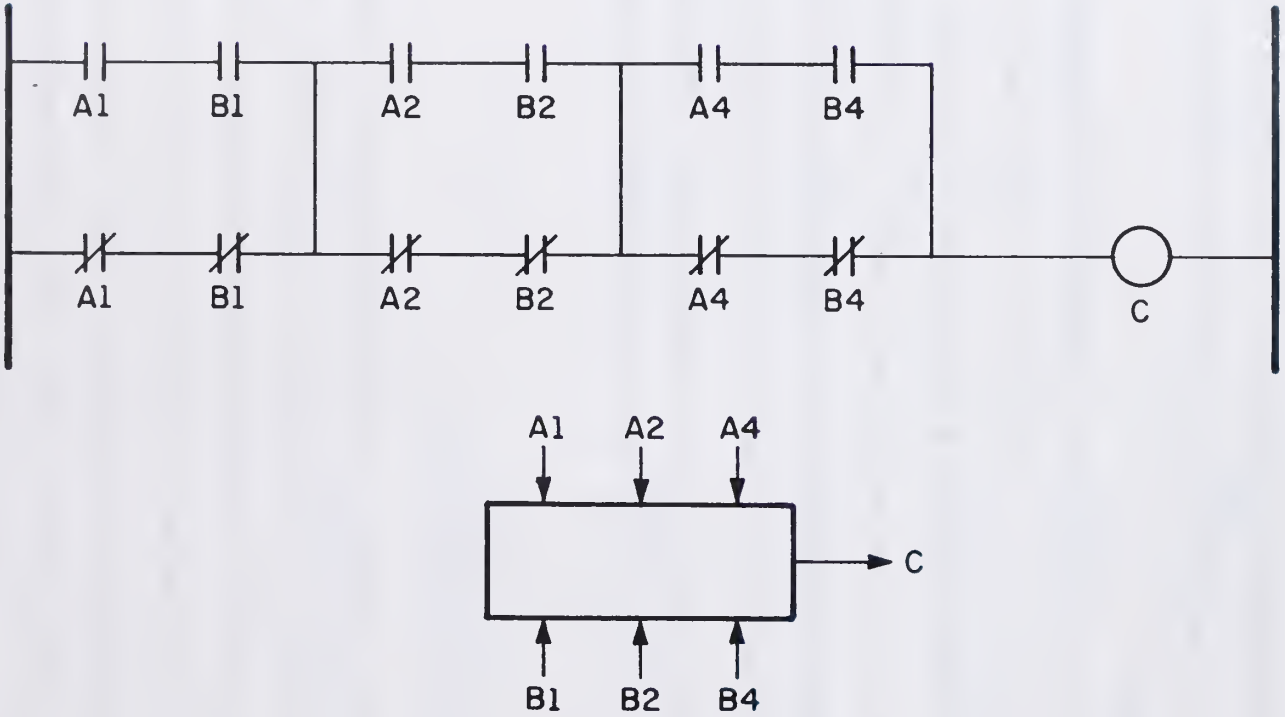
Figure 2. Some RLD Symbols

Essentially a holdover from the days of all relay logic, the RLD seems to have the same persistent appeal as FORTRAN--it is well known; it has been around long enough to be standardized; it can be understood by someone knowledgeable in the area, if not in the particular language; and it is transportable (at least in the system sense) from one machine to another.

This is not to say that RLDs are a perfect form of documentation. Many, perhaps even most, but definitely not all sequential processes lend themselves to description by a relay ladder diagram. For those which do not, some can be forced into such a model, but at the cost of subverting the "language" and obscuring the meaning (see, for example, the 3-bit octal digit comparator described in relay logic in Figure 3). Still, on the whole, the RLD does provide an adequate control description in the majority of cases and it is now the industry standard for documentation [3,4,5].

1.4 Development of Industrial Controllers

In many ways the development of programmable industrial controllers parallels the development of the electronic computer itself, with an appropriate speed-up as befits the pace of modern technology. Within the span of only 40 years the concept of a "computer" has evolved from the mathematical model presented by Alan Turing in 1936 into three generations of machines: the first-generation binary-coded electromechanical computers (Complex Computer, Mark I) and later-model vacuum tube machines (Illiac I), second-generation transistorized machines (IBM 7090/7094),



$$C \leftarrow (A1 = B1) \wedge (A2 = B2) \wedge (A4 = B4)$$

Figure 3. Octal Digit Comparator

and third generation systems exploiting monolithic integrated circuitry (IBM 370, CDC CYBER 175).

Similarly, the last seven years have seen the introduction of a programmable process controller (1969) and its transition from a first-generation implementation in small-scale-integration, high-noise-immunity diode-transistor logic, through a second-generation appearance using predominately medium- and large-scale-integration chips of CMOS technology and designed for an emerging market of original equipment manufacturers, and moving now into a third-generation of controllers whose central intelligence is derived from a microprocessor.

A review of the currently available programmable controllers emphasizes the fact that we are now in a transition period between second and third generation machines. The vast majority of available systems are based on second-generation hardware and only a few have yet exploited the potential of microprocessor-based control units. As shown in a later section, the technology in use is of importance not just because one can claim to have used the most modern technology available, but rather because the difference in second and third generation hardware has such a dramatic impact on the quality of software provided for a given system. Surveys have shown that a potential user is much more concerned with the quality and quantity of software support provided than he is with knowing what hardware technology is used inside his "black box" (controller). The claim of this thesis is that the introduction of microprocessor technology into industrial controllers will result in the most extensive, most comprehensive, most versatile software support yet developed for industrial controllers.

Chapter 2 describes in more detail the development of system software over the last 7 years.

1.5 Choice of Hardware for a New Controller

With regard to the rationale presented thus far, it might appear that a minicomputer would be the logical choice for the computational element of a new controller. Indeed, one major manufacturer (DEC) introduced a minicomputer-based controller in 1969. The arguments in favor of a minicomputer-based system generally center around the availability of minicomputers, the expanding number of manufacturers of both CPUs and peripherals, increasing CPU speeds, and versatile instruction sets. Often not mentioned, however, are the concurrent disadvantages which may be severe. Chief among them are:

1. Size. While minicomputers are not large, they are intermediate in terms of space requirements. Size is a serious consideration when the control equipment is to be either rack mounted or enclosed in cabinets. Heat dissipation from most minis is sufficient to require forced ventilation, which in turn prohibits full environmental isolation. It is imperative that the controller, by virtue of rugged design and adequate enclosure, be made resistant to hostile environments. Oil mists, iron and carbon particulates in the air, very high or very low temperatures, high humidity, and physical shock are common in industrial applications.

2. Cost. The cost of a minicomputer is deceptive, since it represents only a fraction of the total cost of a control system. The

necessary modifications and add-ons (to enhance noise immunity, provide signal conditioning for inputs and outputs, add peripherals for programming, debugging, and monitoring) significantly increase the investment in system hardware. In addition, a portion of the CPU cost buys an extensive instruction set which may represent overkill for the complexity level of the problem being solved.

3. Reliability. LSI technology has done a great deal to increase the reliability of minicomputer components. Yet failures do occur, and when they do the complexity of minicomputer systems usually dictates that a trained computer engineer be called to troubleshoot the system. While system modularity succeeded in making it more cost effective to replace a chip or a whole PC board than to attempt a repair, determining which chip or which board to replace is a big problem and remains beyond the talents of the plant electrician. Owning one controller, or even a few controllers, for a small process or plant is not a sufficient investment to justify a resident computer engineer. In this situation the whole plant is at the mercy of the main-frame manufacturer's field engineers.

4. Noise Immunity. One of the most serious hardware problems is a minicomputer's natural susceptibility to electrical noise. As a rule, the higher the speed of the CPU, the more sensitive it is to electrical disturbance. High-noise-immunity-logic (HNIL), with its excellent tolerance for electrical noise, is a slow speed technology; CMOS, TTL, and Schottky bipolar technologies offer increasing speed but decreasing noise rejection. The physical size of minicomputers (as opposed to, say,

microprocessors), coupled with their noise sensitivity, makes adequate external shielding more costly and less effective.

5. Software. Even assuming that the hardware problems can be overcome, by far the most serious argument against available minicomputers, from the point of view of the end user, is the inadequate software provided for the class of problems being solved. While almost every minicomputer manufacturer is now providing software support such as assemblers, emulators, and perhaps even compilers for BASIC or FORTRAN, these programming tools have proved inadequate for process control problems [7,8]. A point often overlooked is that the average process control engineer neither is nor wants to be a computer programmer. His concern, like that of any specialist, is that he be furnished with the tools necessary for helping him solve his control problem and achieve its implementation quickly, reliably, with an absolute minimum of errors, and with maximal assistance when system debugging and/or tuning are required. While it may be argued that at some future time all process control engineers will become computer programmers and will be comfortable with computer languages and programming techniques, that time is definitely not now; in the meantime, control systems designers are compelled to pay maximum attention to human engineering principles and to spare no resource toward making the programmer's task simple, reliable, and error-free.

1.6 Benefits of a Microprocessor

The choice of a microprocessor as the central logic element not only realizes the full benefits of modern technology but also provides

specific advantages over minicomputers in each of the areas discussed in the previous section.

1. Size. The size advantage is obvious, since microcomputers readily lend themselves to applications where small size is of concern. Heat dissipation is low enough to allow convection cooling and permits installation in contaminated environments requiring total enclosure of the controller.

2. Cost. While the cost of the CPU still represents only a fraction of the total system cost, a $25 microprocessor compares very favorably with a $300 to $1000 minicomputer. Even after adding the necessary memories and I/O ports, the total hardware investment represents one of the least expensive control systems available.

3. Reliability. By having the entire CPU on one chip, and the CPU, memories and I/O ports on one PC board, the reliability of this system is greatly enhanced. Diagnostic tools are designed into both the hardware and software to permit troubleshooting in the field. Replacement of a defective chip or even the entire processor board can be accomplished in minutes even by persons with limited hardware experience.

4. Noise Immunity. The specific techniques used to gain noise immunity are beyond the scope of this thesis. However, the major contributions to noise rejection include photoisolated inputs, transformer-coupled outputs, multiple ground returns in the PC board, and last but not least, the small size of the microprocessor itself and its associated components which make physical shielding of the entire controller a cost-effective deterrent to radio frequency interference.

5. Software. In adherence to human engineering principles, this thesis places heavy emphasis on the quality, utility, and versatility of the software packages provided, particularly the choice and implementation of the programming langauge. Obviously, there is nothing in the microprocessor software which could not have been accomplished on minicomputers or IBM 370s. What is unique is that the microprocessor-based hardware gives the system designer greatly increased flexibility as he decides which functions to implement in hardware and which in software. The added flexibility of a software-based controller, rather than a hardware-based controller, is sufficiently advantageous to justify the extensive investment in microprocessor software.

1.7 Defining "Satisfactory Software"

Our answer to the question "What is satisfactory software for an industrial controller?" has profound effects on every aspect of the controller's design. A key design philosophy has been that by developing the hardware and the software simultaneously, rather than retrofitting either to the other, one may realize the ultimate in cooperation between these two different (and sometimes antagonistic) worlds. Thus it was of paramount importance that the needs of the end user be considered before the first byte of software was coded.

The most important software decision is the definition of the control language in which the user describes (programs) his particular control sequence. What should be the nature of this language? Microprocessor assembler code? BASIC? FORTRAN? A special Boolean language

invented just for process control? None of the above? Since this is such a critical decision, let us first establish our goals for the control language. After all, the user doesn't care how many operating systems or compilers the system uses as long as the programming task is made as simple as possible. Chapter 2 discusses some existing control languages and their problems.

For now, we examine our goals in the context of what the user brings to the programming system (usually a relay ladder diagram and perhaps a set of Boolean equations taken from the RLD) and attempt to determine what the user expects from the programming system.

Conceptual Simplicity

The end user is a process control engineer or perhaps a plant electrician. A method of programming which utilizes--indeed, which capitalizes on--the intuition of this new class of programmers will go a long way toward selling itself; at the same time it will reduce the frequency of programming errors due to unfamiliarity with style or syntax.

The programming methodology must speak the language of the user, and that language is relays, solenoids, series and parallel connections, etc. Solution: Use a programming language oriented toward conventional relay symbols. Make it a graphic language to insure maximum similarity between the programming language and the source document, the RLD.

Usable by Nonprogrammers

Generally speaking, our prime users (process control engineers)

have not been, are not now, and in the future do not wish to be computer programmers.

They do want to accomplish the "programming task" without resorting to unfamiliar languages in which consideration of syntax, semantics, and strange formatting rules continually divert the programmer's attention from his basic problem. Ideally, the programmer should not have to program, or at least he should not feel like he is programming. He should be copying his RLD from one medium to another.

Solution: Program interactively; prompt the user at every step; design the language so that syntax errors are hard to make; when a syntax error does occur, force its immediate correction.

Easy to Program/Easy to Edit

Any programmer is irritated when he can't easily enter a prepared program or modify an existing one. If entering a program requires very much special or expensive equipment, or long-distance telephone connections to a central computer facility, or manual programming of hundreds (and sometimes thousands) of memory locations via thumbwheel switches and pushbuttons, the utility of the programming system, however clever, is much degraded. Likewise, if simple changes to an almost-working program (like adding a series contact to a horizontal line) means manually reprogramming the memory from that point on, the user is sure to be frustrated.

Solution: Make the programming unit (hereafter called a program loader) a single, self-contained, separate piece of equipment from the

controller; the program loader should be usable on-line for monitoring, testing, and quickie fixes, or off-line for program development; make it connectable to a controller by a single plug-in cable; make its software compatible with all models of the controller so that one programming unit can service many controllers for program entry, editing, and monitoring.

Versatility

From the point of view of the end user, it should be just as easy to control a timer, counter, shift register, or stack as it is to control a real world output. There should be no funny changes in programming style just because the controlled device is not a physical output (i.e., semantics should not affect syntax).

Solution: Designate one set of symbols to use as outputs for all types of variables and devices; use addressing to indicate exactly what type of output and which of many similar outputs is being controlled.

Free Format

There should be a minimum number of restrictions on the exact format of the input. As far as practical, we should not make the user recast his input to fit a set of formatting rules.

Solution: Use the very minimum number of syntax rules necessary to guarantee that the user is generating a meaningful, logical, solvable, syntax-error-free program.

Generate Standard Documentation

When programming is finished, the system must be able to produce a relay ladder diagram which documents the control program. The form of the RLD must comply with industry standards for hard-copy documentation [3,4,5].

Recover the Original RLD from the Internal Code

After the system has been running for a year and its supporting documentation lost, or after extensive undocumented tuning in the field, it is advantageous to have the system reconstruct the original RLD from its internal code. With some effort it is possible to construct from, say, compiled microprocessor code implementing primarily Boolean logic, an RLD which is equivalent (i.e., logically correct but geometrically different) to the original programming document, but recovery of the exact original is much more useful, even though it implies more encoding in the internal code.

Of these seven goals, the last (recovery of the exact RLD from the internal code) is the most restrictive and has the most impact on the choice of a programming language.

1.8 Choosing the Programming Language

In light of the desired goals for the programming langauge, what makes the most sense? We examine the major possibilities.

Assembler Code

We could require the user to learn the assembler code for the particular microprocessor being used and then provide a resident assembler. While an experienced programmer might be able to examine an RLD, extract its governing set of Boolean equations, and translate them into microprocessor code, this is clearly unreasonable for a nonprogrammer. Also, should some future version of the hardware utilize a different microprocessor, all the users would have to be retrained in another programming language. Of the seven goals, assembler coding satisfies only the requirement that it could be analyzed and translated into an RLD; thus, assembler code is rejected.

FORTRAN

Since a higher-level language seems to be indicated, what about an existing language like FORTRAN? Most would agree that FORTRAN is not conceptually simple and its function not obvious to nonprogrammers. Its ease of programming and editing would depend on the implementation. It is versatile, but its operators are designed more for arithmetic than for Boolean algebra. Translation from Boolean algebra to FORTRAN would be simple, but translation from an original RLD would require the extraction of Boolean equations as a manual and useless intermediate step. An RLD plotter using FORTRAN code as the source input would be possible, but reconstruction of an original RLD is not unless excessive redundancy is introduced.

BASIC

If assembly language is too machine oriented and FORTRAN is too complex, perhaps BASIC is a reasonable compromise. This was an original suggestion until a survey [26] showed BASIC to be still "too computer-ish." Actually, BASIC is not much better than FORTRAN for process control applications; its main differences lie in its simpler syntax and control structures. Its operators do not encourage Boolean operations and the user would still have to generate Boolean equations from his RLD as an intermediate step.

VIPTRAN

Some work has been done toward inventing higher-level programming languages specifically designed for process control [6,9,10]. VIPTRAN and VIPTRAN2 are high-level, FORTRAN-type languages developed by this author (1972-1973) to provide programming ease and flexibility for the VIP series of industrial controllers manufactured by Struthers-Dunn, Inc., Bettendorf, Iowa. The users of this software system were experienced process control engineers who were very comfortable with Boolean algebra and RLDs. VIPTRAN succeeded in providing the advantages of programming in a high-level language, letting the compiler do address assignment and automatic generation of control variables, providing for easy programming and editing on an interactive computer system, and allowing free-format input. Perhaps its greatest utility was in the use of an auxiliary plotter package which used the compiler-generated machine

code to generate a relay ladder diagram for documentation. If the compiler's input set of Boolean equations had been hand-translated from an RLD, the plotter's output would in all cases be equivalent, but not necessarily identical, to the original (see Figure 4). Others [6,9,11] have also designed special languages to bridge this communication gap, but to date none accept arbitrary free-format input or recreate the original RLD from the internal code.

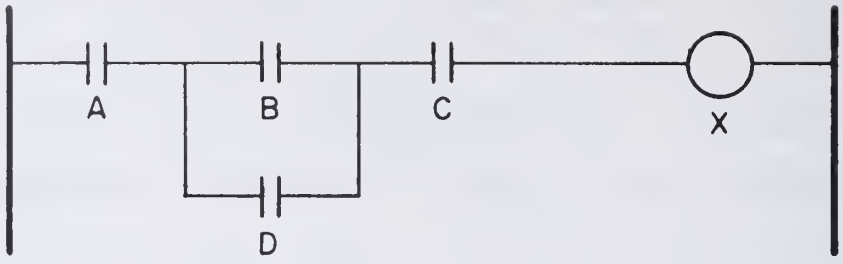
Relay Ladder Diagram

The idea of using the original RLD itself as the source input has been discussed [12,16] but remained unimplemented until recently. Even so, the existing implementations [12,17,19,22] attach a rigorous set of formatting rules to the RLD so that the difficulty of translation (some via hardware and some via software) is minimized. While this approach does simplify the amount and complexity of system software, it needlessly restricts the programmer's freedom of expression by requiring the user to reformat (redraw) a conceptual RLD to conform to a strict graphic syntax.

Typical restrictions include:

1. Only four or five horizontal elements per line [12,13,14];
2. Only two horizontal lines per output [13];
3. Parallel connections must return to the power line [12,13, 14,15,21];
4. Only one output per display screen [12,13,14,15,18,19,22];

Original RLD



VIPTRAN equation

```
X = A * (B + D) * C
```

Compiler output

```
LDA     B       ; load B
OR      D       ; OR with D
AND     A       ; AND with A
AND     C       ; AND with C
STO     X       ; store into X
```

VIPTRAN plotter output

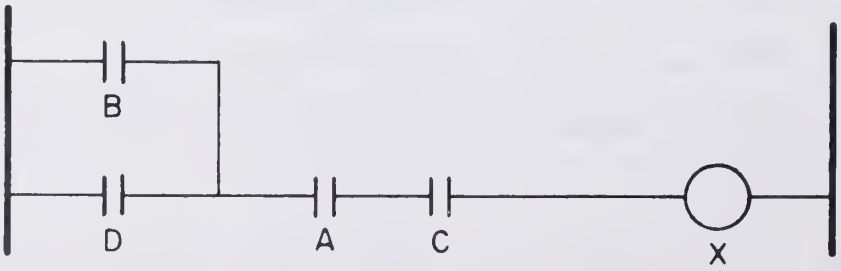


Figure 4. VIPTRAN Coding Sequence

5. Output must be in upper right corner of display [12,13, 14,15,18,19,22]; and

6. Only normally-open outputs allowed [12,13,14,15,18,19,22].

Careful design and more elaborate software can remove most of the formatting restrictions, including all six mentioned above.

As for compatibility with the system software goals (Section 1.7), the RLD can meet all seven requirements. By designing a graphically-programmed, microprocessor-based program loader with a CRT display and a keyboard containing pushbuttons for all necessary relay symbols, cursor controls, editing facilities, and mode selection for programming, editing, and monitoring, we can implement a programming "language" whose rules, from the point of view of the user, are hardly more difficult to learn than those for operating a touch-tone telephone. Significantly, since the input source and the output documentation are the same type (RLDs), the user may rely on visual verification to ascertain whether his input program is in agreement with his intentions.

As for the seven system software goals:

1. Conceptual Simplicity. The symbols used are the relay symbols familiar to a control engineer. His intuition is used to advantage because "programming" the controller only means copying his RLD from paper to CRT via pushbuttons.

2. Usable by Nonprogrammers. Each keypress results in some visible change to the CRT screen. If the change to the screen is correct, the user continues programming via more keypresses; if not, he is in full control and may replace any erroneous symbol or address immediately. The

user has full control of a cursor so that he may program any element on the screen in any order, thus eliminating the strict left-to-right syntax which is both common and annoying [17].

3. Easy to Program/Easy to Edit. In addition to the features described above, a compiled program may be decompiled, displayed, edited, and recompiled in seconds. The editing functions allow addition, deletion, or replacement of any element anywhere in the program. A "search" mode is provided for locating all occurrences of a particular symbol type and address.

4. Versatile. Because addressing determines variable type, it is just as easy to reset a timer or make a counter count up as it is to actuate a motor starter.

5. Free Format. The formatting rules are so few and so simple that the user is almost unrestricted with regard to program entry. This feature points out another unique advantage--if a control system can be drawn on the CRT, it can be solved by the controller.

6. Generates Standard Documentation. Since the RLD is the input, it is self-documenting. When the controller is connected to a program loader, an RS-232C I/O port permits hard-copy documentation of the RLD by a teletype or computer.

7. Recover the Original RLD from the Internal Code. The design of the graphic translator insures that the exact geometry of the input is preserved by the internal code and is recoverable at any time.

1.9 Overview

Having established that an interactive, graphically-programmed,

microprocessor-based industrial controller would be of significant value when used in the control of machines and processes of intermediate complexity, the remainder of this thesis deals with the design and implementation of just such a system (controller and program loader).

Chapter 2 provides a brief history concerning the development of controller software from 1969-1976; Chapter 3 discusses techniques for translating RLDs into usable internal codes; Chapter 4 elaborates on the hardware and software requirements of the controller itself; Chapter 5 provides the equivalent information for the program loader; Chapter 6 deals with the logic design and software problems encountered when connecting dual microprocessor systems; and Chapter 7 summarizes and generalizes the results of this effort and identifies some areas which would profit from additional research.

2. DEVELOPMENT OF INDUSTRIAL CONTROLLER SOFTWARE (1969-1976)

2.1 Introduction

Since the introduction of the first programmable industrial controllers in 1969 by DEC [32], Allen-Bradley [20], Modicon [12], and Struthers-Dunn [1], approximately 25 additional manufacturers have joined in the effort to produce programmable control systems. Each of these systems accomplishes the same basic task--the replacement of relays with programmable, solid state logic--with a resultant increase in both system flexibility and reliability. While controller hardware has generally kept pace with computer hardware technology, this fact is less visible to the user than the quality and quantity of software provided. As a result, much effort has been recently expended to improve the man-machine interface at the system software level.

It is pointless to examine every industrial controller available since many show significant similarities. What is useful is a brief examination of three major systems, each of whose programming system was a significant milestone in the development of controller software. In this analysis we see the same progression from machine code to assembly language to high level languages that marked the development of computer science, and, in fact, it was the development of the computer sciences which made the "software revolution" in industrial controllers possible.

2.2 Allen-Bradley PMC (1971)

The PMC (Programmable Matrix Controller) is designed for on-site programming and memory loading. The control program is written using 6 basic instructions and then entered into the read-only-memory using an auxiliary "Memory Loader" unit. The instruction set permits direct implementation of Boolean OR statements in which each branch of the OR may contain any number of AND terms. The number of OR terms in any one Boolean equation is limited only by the physical memory size.

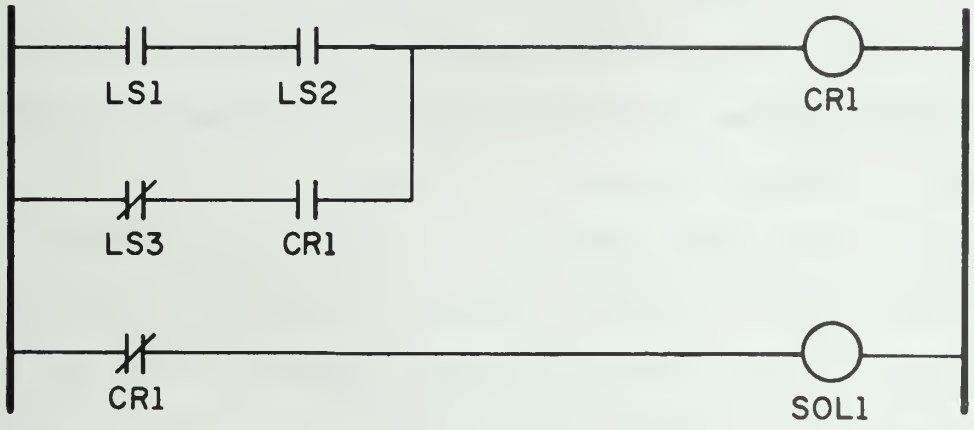
The instruction set includes:

XIC (eXamine Input Closed)	Determine if the specified input is in the closed state at the time of examination.
XIO (eXamine Input Open)	Determine if the specified input is in the open state at the time of examination.
XOE (eXamine Output Energized)	Determine if the specified output is energized at the time of examination.
XOD (eXamine Output De-energized)	Determine if the specified output is de-energized at the time of examination.
BRT (BRanch Test)	This instruction must follow the last examine of each branch of an OR statement, except the last branch.
SET	This instruction must follow the

last examine instruction of an AND statement, or follow the last examine instruction of the last branch of an OR statement. If the required examine conditions are met, the output associated with the SET instruction will be energized; else the output will be de-energized.

Two no-operations are also found in the instruction set. One of the NOPs, with an instruction code of all zeroes, can be used to reserve memory in anticipation of future changes or expansion (due to the nature of the ROM, a zero bit can be later changed to a one, but not vice-versa).

The programming technique is best illustrated by following the steps necessary to implement a very simple control problem. The English language statement of the problem is: Control relay #1 is to be energized if and only if either limit switch #1 is closed and limit switch #2 is closed, or if limit switch #3 is open and control relay #1 is already energized. Solenoid #1 is to be energized if control relay #1 is de-energized and is to be de-energized when control relay #1 is energized. From this conceptualization the user would document his control logic with a simple relay ladder diagram such as this one.



From the RLD the user would now manually extract the equivalent Boolean equations:

$$CR1 = (LS1 \wedge LS2) \vee (\overline{LS3} \wedge CR1)$$

$$SOL1 = \overline{CR1}$$

From the Boolean equations the user now generates the control program in pseudo-assembly language ("pseudo" because there is no assembler!).

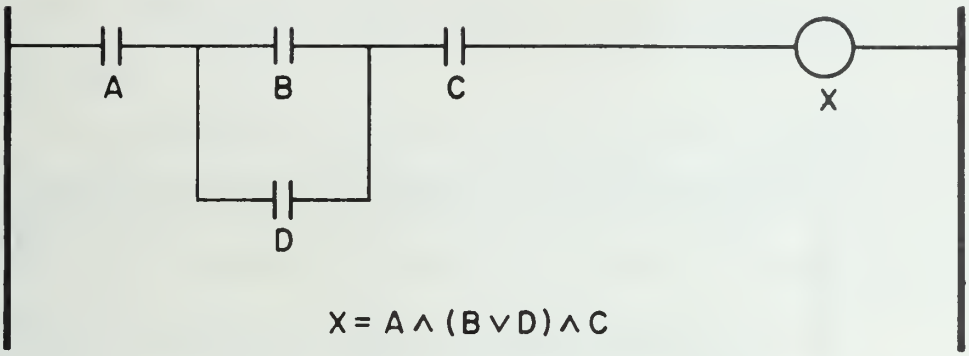
XIC	LS1	is limit switch #1 closed?
XIC	LS2	AND is limit switch #2 closed?
BRT		OR
XIO	LS3	is limit switch #3 open?
XOE	CR1	AND is control relay #1 energized?
SET	CR1	energize CR1 if either OR term is true; else de-energize CR1.
XOD	CR1	is CR1 de-energized?
SET	SOL1	energize SOL1 if true; else de-energize SOL1.

The symbolic names used above (LS1, CR1, etc.) are now manually translated into specific I/O addresses as required by the particular installation (e.g., LS1 might be physically connected to input terminal #7). To program the controller, a memory module (64 words of ROM) is placed in the Memory Loader unit. To insert an instruction, the memory address (00-63) is set on a two-digit thumbwheel switch, the I/O address (if any) is set on another two-digit thumbwheel switch, and a pushbutton corresponding to the desired instruction (one of eight) is depressed. These steps are repeated for each of the 64 instructions in a ROM memory module.

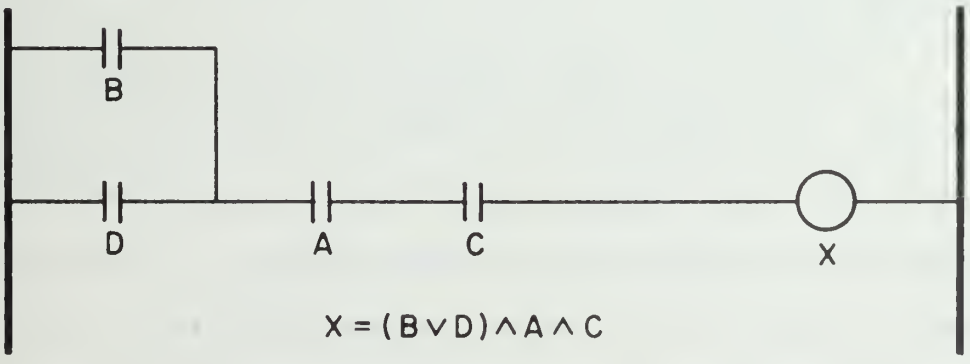
The contents of any memory element can be verified after programming by setting the memory address switch to the address of the memory word which is to be interrogated, and by then observing which pushbutton is illuminated and examining indicator lights to determine the associated I/O address.

This system is typical of those of its time. It is an all-hardware system with no software support at any point in the programming process. The "programming language" leaves much to be desired since it forces the Boolean coding to conform to disjunctive normal form (OR of AND terms). In terms of the equivalent relay ladder diagram, this restriction is equivalent to requiring that all OR branches (parallel contacts) must return to the power line. Thus, any RLD not initially drawn in this fashion would have to be redrawn and its Boolean equations restated to conform to this rather stringent requirement.

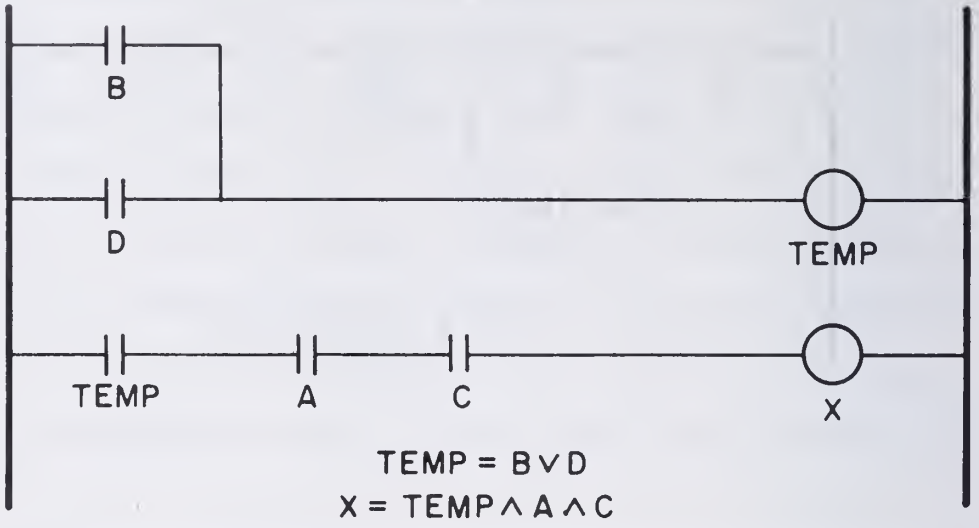
For example, even this simple RLD does not fit the requirements.



It must be manually translated into either this form



or, by adding an additional variable, into this form.



The manual assignment of addresses to variable names is another tedious task if explicit assignments are not already required by the particular application.

Editing an existing program is likewise a tedious and undesirable task. Since each bit in the ROM is initially zero and can only be changed into a one, editing is restricted to those changes which can be implemented by changing existing zeroes into ones. If the desired change cannot be obtained in this manner, the only alternative is to discard the current memory module and program a new one from the beginning. This situation encourages a user who is unsure of his program to leave some all-zero NOPs embedded in his program; the NOPs can then be left as is or programmed into useful instructions. Erasure can be accomplished only

by turning an existing instruction into the all-ones NOP. The net result is wasted memory space and decreased execution speed, even in production programs.

No system facility exists to permit dynamic monitoring of inputs or outputs by either the Memory Loader unit or the PMC itself. Rather, the PMC is prewired to allow for an optional, custom interface with an external computer if monitoring is desired.

In summary, the lack of any system software makes programming, editing, and monitoring completely manual operations, each one accomplished by individual hardware boxes attached to the basic PMC. The requirement of programming in disjunctive normal form forces the redrawing of RLDs, recasting of Boolean equations, and manual introduction of temporary variables, all of which serve to confuse, confound, and irritate the user. The primitive editing facility gives the user small margin for error and induces him to waste memory in an effort to save editing time.

2.3 Struthers-Dunn VIP with VIPTRAN (1972)

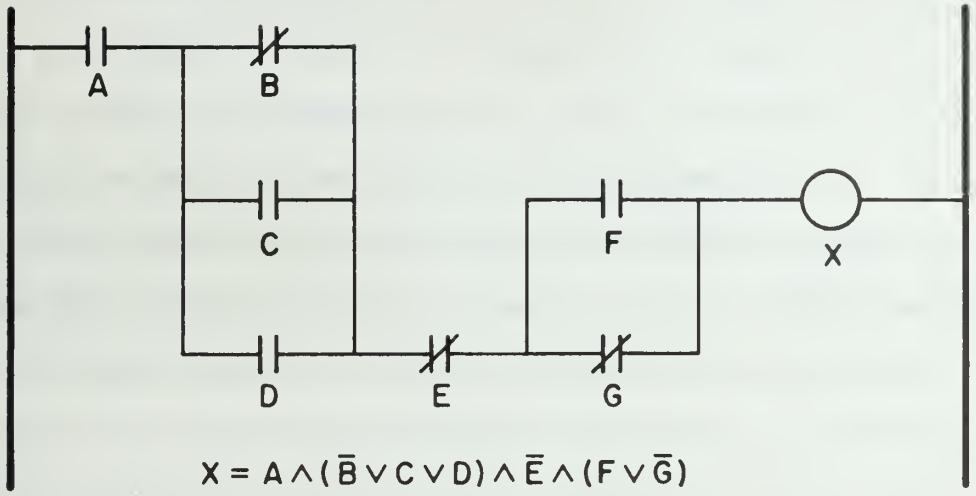
The original Struthers-Dunn VIP, introduced in 1969, utilizes a programming language similar to that described for the PMC. The chief differences are the mnemonic instruction names (designed specifically for programming Boolean equations) and the number of instructions available.

The VIP architecture uses a one-bit accumulator and single-address instructions. Its instruction set includes:

LDA	X	load the accumlator with X
LDAC	X	load the accumulator with the complement of X
AND	X	AND the accumulator with X
ANDC	X	AND the accumulator with the complement of X
OR	X	OR the accumulator with X
ORC	X	OR the accumulator with the complement of X
STO	X	store the accumulator into X

as well as 8 auxiliary control instructions.

Programming from an RLD proceeds as before; first, manually translate the RLD into Boolean equations.



Proceeding by hand, one can now transform the Boolean equations into their assembly language equivalent, resulting in a nonunique implementation such as:

LDAC	B	load complement of B
OR	C	OR with C
OR	D	OR with D
STO	temp1	save $(\overline{B} \vee C \vee D)$ in a temporary variable
LDA	F	load F
ORC	G	OR with complement of G
STO	temp2	save $(F \vee \overline{G})$ in another temporary variable
LDA	A	load A

AND	temp1	AND with $(\overline{B} \vee C \vee D)$
ANDC	E	AND with complement of E
AND	temp2	AND with $(F \vee \overline{G})$
STO	X	save final result in X

The production of space-optimal code depends entirely on the facility of the user, his familiarity with the instruction set, and his understanding of basic assembly language programming and optimization techniques. However, due to the nature of the instruction set, the user is no longer constrained to any particular programming style (disjunctive normal form or otherwise). Yet, the one-accumulator architecture does force the frequent introduction of temporary variables whenever the logical expressions are nested more than one level deep.

Obviously, the use of Boolean equations to represent the logic of relay ladder diagrams was not new with VIPTRAN: Boolean algebra was well established as a method of documentation for relay logic. In addition, the design of the original VIP assembly language based on Boolean operands and the mechanism for translating RLDs into VIP assembler code (albeit by hand) had been suggested by T. A. Murrell as early as 1968 and implemented by Struthers-Dunn, Inc., in 1969. Of course, the experienced VIP coder could often omit the Boolean equations and code directly from the RLD, although the beginner was encouraged to write down the intermediate Boolean equations.

It was this necessity for hand tanslations which encouraged the development of VIPTRAN [27] (1972) and VIPTRAN2 [28] (1973), two programming languages and cross-compilers designed especially for process

control problems. The introduction of the VIPTRANs was a valuable achievement since it not only guaranteed error-free code but also eliminated hand compilation of Boolean equations or hand assembly of VIP code. While VIPTRAN did introduce the necessity of explicitly providing Boolean equations which was, in a sense, previously optional, it compensated for this additional intermediate step by using those same Boolean equations as its input source text. By so doing VIPTRAN avoided the invention of yet another high level programming language, and instead arranged the syntax of the language so that the user could program in a familiar language--that of Boolean equations.

As a software system, VIPTRAN has the capability of performing the entire programming task from compiling the source code to programming ROM and PROM memory chips. VIPTRAN's main features include:

1. User-specified or automatic (or both) assignment of variable names to real world I/O terminals, properly separated by voltage and type;
2. Optimal allocation of user and system variables to conserve hardware;
3. Automatic choice of the proper VIP model based on the complexity of the source code;
4. Automatic selection and mapping of the proper type and quantity of hardware I/O modules;
5. Optional sorting of equations to identify and either eliminate or reduce equation interdependencies (e.g., an equation calculating the value of X should precede equations using X as an operand);

6. Optimized compilation of VIPTRAN source code, automatic generation and allocation of temporary variables whenever necessary, and generation of VIP machine code;
7. Provides a data type for on-delay and off-delay timers as well as generic time delay functions;
8. VIPTRAN2 provides a significant degree of both syntactic and semantic error detection (and repair when possible);
9. Provides for programming ROMs and PROMs directly from the output machine code; and
10. Generates an equivalent RLD directly from VIP machine code.

The ability to reconstruct the RLD is useful to the engineer because it provides immediate, understandable documentation for the control scheme, aids system startup and debugging, and enhances the ability to visually verify program correctness.

Figure 5a shows an original RLD and its Boolean equation; Figure 5b is the VIPTRAN2 program which the user would write; the VIPTRAN2 compiler output is shown in Figure 5c; Figure 5d is the RLD as reconstructed from the machine code output.

In summary, VIPTRAN was a significant step toward automating the programming process. While the user was required to manually translate his RLD into Boolean equations, the programming system was capable of accepting those Boolean equations as the source text and performing all the remaining programming tasks.

The advantages of VIPTRAN programming are numerous, especially when compared to assembly language coding, and the system was readily

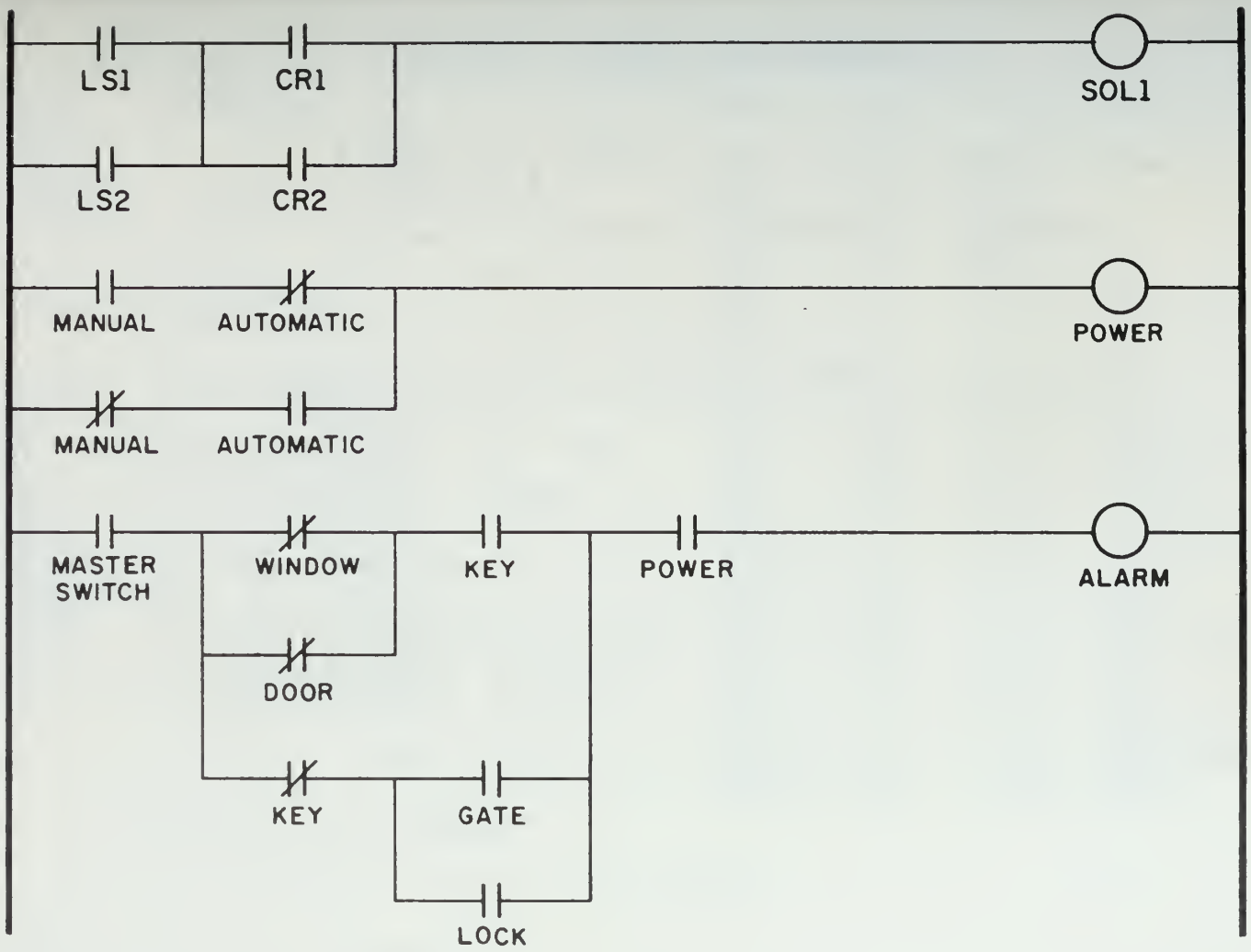


Figure 5a. Original Relay Ladder Diagram

```
VIPTRAN2 COMPILER, VERSION 2.00
DATE: 09/25/76 TIME: 21.46.16.880

<OPTIONS>  SOURCE NOSORT NOPRINTSORT CODE NOSORTFAIL MAP NOTTY NOSCRATCHPAD SKIP0

CARD#  STMT#      SOURCE STATEMENT
    1      0     C
    2      0     C
    3      0     C PROGRAMMING EXAMPLE FOR VIPTRAN2.
    4      0     C
    5      0     C
    6      0     C DECLARE THE INPUTS.
    7      0     C
    8      0     C
    9      0     C INPUT6.   WINDOW  DOOR  KEY  LS1  LS2
   10      0     C
   11      0     C INPUT12.  GATE    LOCK
   12      0     C
   13      0     C INPUT48.  MANUAL  AUTOMATIC
   14      0     C
   15      0     C INPUT120. MASTERSWITCH  POWER
   16      0     C
   17      0     C
   18      0     C DECLARE THE OUTPUTS.
   19      0     C
   20      0     C
   21      0     C OUTPUTAC. SOL1   POWER
   22      0     C
   23      0     C OUTPUTDC. ALARM  CR1  CR2
   24      0     C
   25      0     C
   26      0     C THE CONTROL PROGRAM IN THE FORM OF BOOLEAN EQUATIONS
   27      0     C AS TAKEN FROM THE RELAY LADDER DIAGRAM.
   28      0     C
   29      0     C PROGRAM.
   30      1     C
   31      1     C SOL1  =   (LS1 + LS2) * (CR1 + CR2)
   32      2     C
   33      2     C POWER =   (MANUAL * /AUTOMATIC) + (/MANUAL * AUTOMATIC)
   34      3     C
   35      3     C ALARM =   MASTERSWITCH *
   36      3     >           ( (/WINDOW + /DOOR) * KEY + /KEY * (GATE + LOCK)) *
   37      3     >           POWER
   38      4     C
   39      4     $TRANSLATE
```

Figure 5b. VIPTRAN2 Source Program

VIP CARD MAP

TYPE	ADDRESSES		
INPUT6	0	->	17
INPUT12	20	->	37
INPUT48	40	->	57
INPUT120	60	->	77
OUTPUTAC	100	->	117
OUTPUTDC	120	->	137

VARIABLE MAP

NAME	ADDRESS	TYPE
WINDOW	1	INPUT6
DOOR	2	INPUT6
KEY	3	INPUT6
LS1	4	INPUT6
LS2	5	INPUT6
GATE	20	INPUT12
LOCK	21	INPUT12
MANUAL	40	INPUT48
AUTOMATIC	41	INPUT48
MASTERSWITCH	60	INPUT120
POWER	100	OUTPUTAC
SOL1	101	OUTPUTAC
ALARM	120	OUTPUTDC
CR1	121	OUTPUTDC
CR2	122	OUTPUTDC

VIP MACHINE CODE

WORD#	INSTR	ADDR	VARIABLE NAME
0001	LDA	000	
0002	AUX	776	
0003	AUX	775	
0004	AUX	774	
0005	STO	000	
0006	LDA	004	LS1
0007	OR	005	LS2
0010	STO	102	TEMP0102
0011	LDA	121	CR1
0012	OR	122	CR2
0013	AND	102	TEMP0102
0014	STO	101	SOL1
0015	LDA	040	MANUAL
0016	ANDC	041	AUTOMATIC
0017	STO	102	TEMP0102
0020	LDAC	040	MANUAL
0021	AND	041	AUTOMATIC
0022	OR	102	TEMP0102
0023	STO	100	POWER
0024	LDAC	001	WINDOW
0025	ORC	002	DOOR
0026	AND	003	KEY
0027	STO	102	TEMP0102
0030	LDA	020	GATE
0031	OR	021	LOCK
0032	ANDC	003	KEY
0033	OR	102	TEMP0102
0034	AND	060	MASTERSWITCH
0035	AND	100	POWER
0036	STO	120	ALARM

Figure 5c. VIPTRAN2 Address Assignment and Object Code

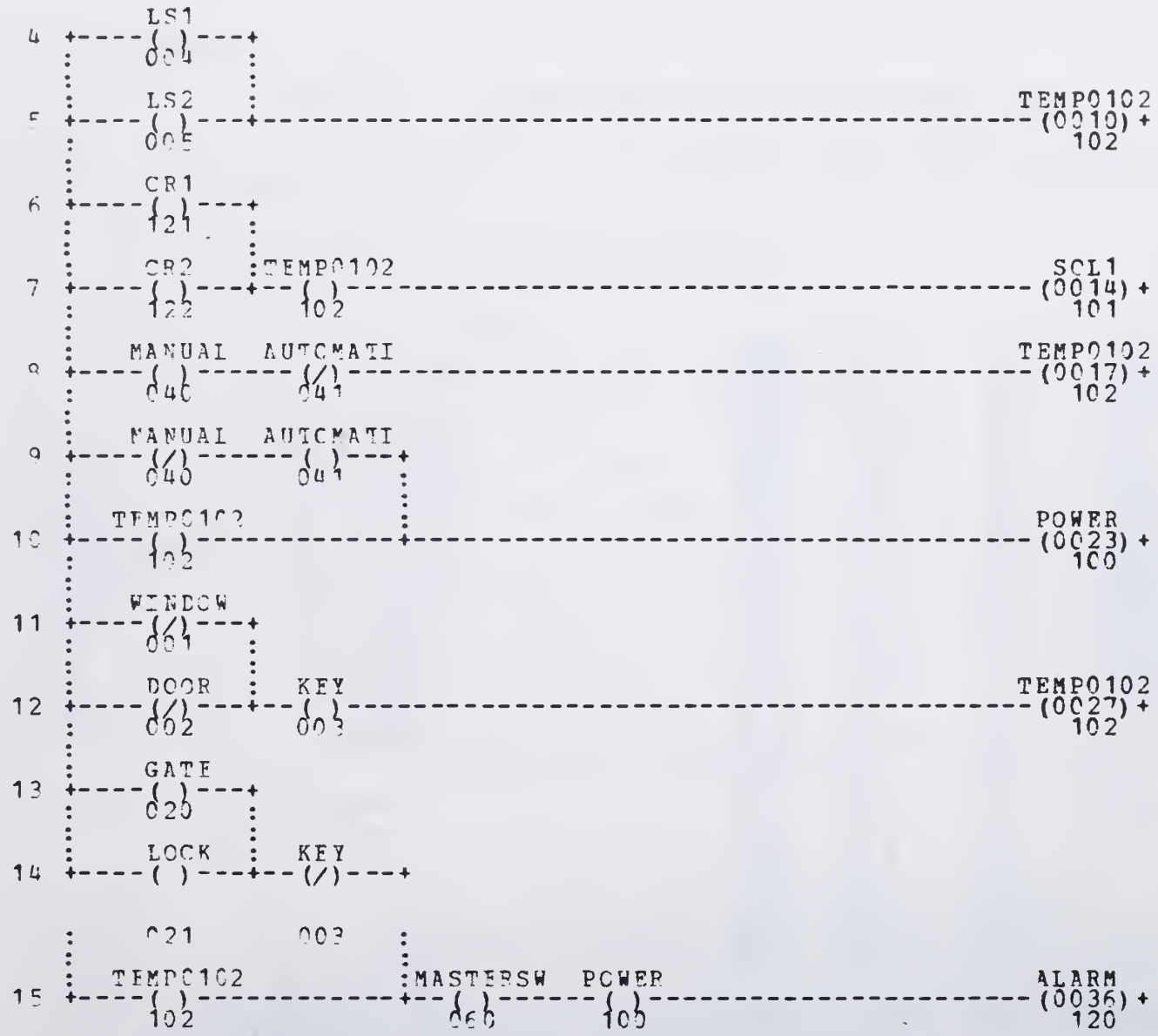


Figure 5d. Plotter Output

accepted by control engineers, the majority of whom were comfortable with Boolean algebra. The prime disadvantages were the cross-complier's dependency on a separate, large-scale computing system and VIPTRAN's inability to regenerate the original topology of the RLD.

2.4 DEC Industrial 14 with VT-14 (1973)

Digital Equipment Corporation marketed the original PDP-14 process controller in 1969; its programming, editing, monitoring, and debugging were all dependent upon having an attached PDP-8. The software for the PDP-14 was exactly that provided for the PDP-8. When system development was completed, the user sent his completed PDP-8 control program to DEC, who returned a custom-manufactured braided-wire memory for the PDP-14. In 1973, DEC announced a new controller, the Industrial 14 with VT-14 software [17]. The VT-14 system allows the user to specify his control logic using a set of relay symbols input via pushbuttons and displayed on a CRT. Figure 6 shows the VT-14 programming terminal and Figure 7 illustrates its keyboard.

The VT-14 defines a programming matrix of 8 rows and 10 columns on the CRT. Each matrix position may be a relay element, a branch point, or blank. Only one output is allowed per screen display and that output is constrained to be in the upper right-hand corner of the screen.

The allowable matrix elements are:

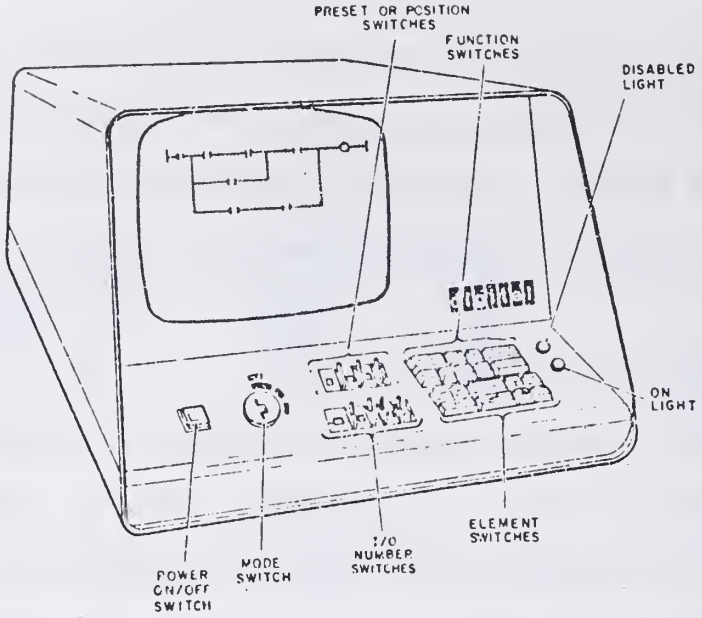


Figure 6. VT-14 Programming Terminal

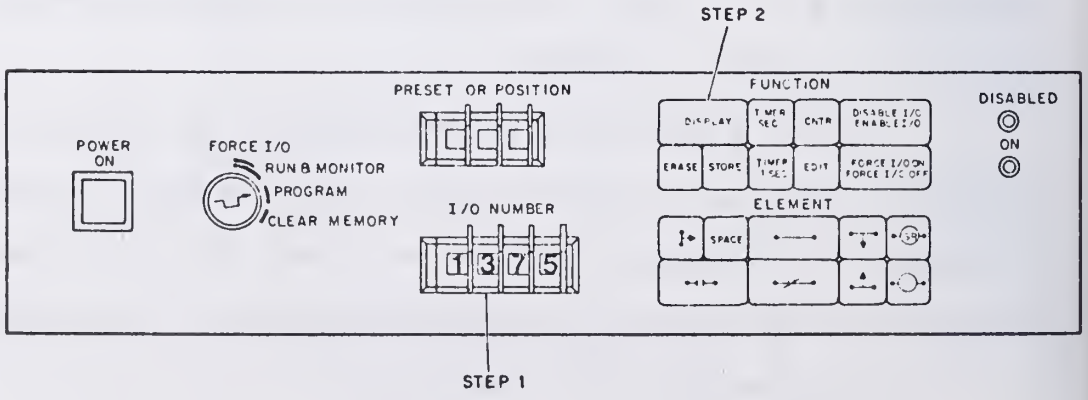
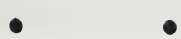
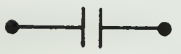
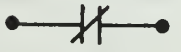


Figure 7. VT-14 Keyboard

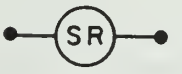

blank


horizontal connection


normally open relay


normally closed relay


output



shift register


begin down branch


end down branch

In addition to the pushbuttons for relay elements, the VT-14 provides a 4-digit thumbwheel switch for specifying I/O addresses and another 3-digit thumbwheel switch used during program editing for specifying which matrix position is to be altered.

The format of the input is subject to the following restrictions:

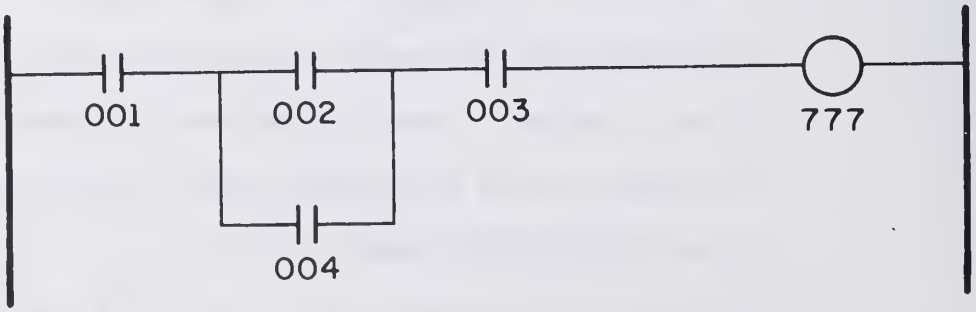
1. The circuit is entered left to right and top to bottom. Each horizontal path must be completed before the next line is started. Having passed over an element, it cannot be changed during programming mode; it can only be altered by switching to edit mode.
2. The one and only output coil () is always

the last element of the first row and can appear in no other location.

3. When one horizontal line has been completed, the next one is started by pressing the ⊢→ symbol.
4. Branches may be used anywhere within a circuit by positioning a down branch symbol (⊤↓) to initiate the branch and an up branch symbol (↑⊥) to reconnect the branch.
5. Branches must be aligned vertically. Either the space key or the horizontal connection (●——●) must be used to lengthen a short line so that it matches the length of the longest line of a parallel contact.

When the programming of one page (one display screen) is completed, the STORE key is depressed and the circuit is recorded in the controller's memory.

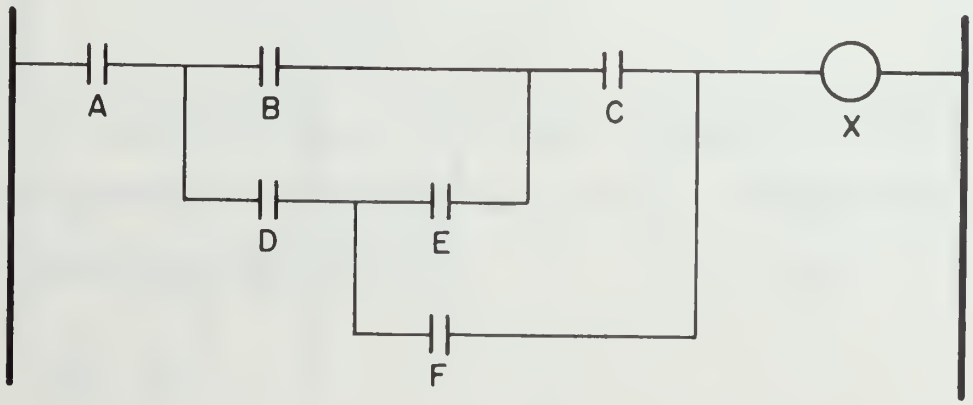
Editing a program is more complicated than programming because the user must specifically identify the matrix position he wishes to change by its one-digit row (0 - 7) and column (0 - 9) number. For instance, suppose the following RLD had been input:



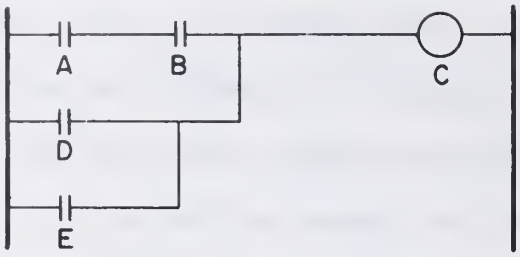
Changing the normally open contact at address 003 to a normally closed contact requires editing the third relay symbol in the first line. Counting across the first line, the first element is a relay, the second a branch point, the third a relay, the fourth a branch point, and the fifth is the element in question. The fifth element on line one is in matrix position (row,column) = (0,4) so 004 is set on the 3-digit thumbwheel switch and the EDIT pushbutton depressed. Now the 3-digit I/O address (003) is dialed on the 4-digit I/O number switch (0003) and the pushbutton depicting a normally closed relay is depressed. At this point the screen will display the corrected element. Pressing STORE again replaces the old page in memory with the current, corrected page shown on the CRT. Translation from the RLD into internal code requires approximately 20 seconds [17].

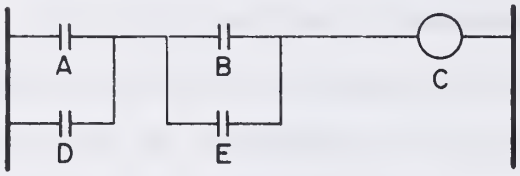
The formatting rules are fairly strict and prohibit arbitrarily formatted input. RLDs which do not conform to the formatting rules must be redrawn, as shown by the examples on the following page.

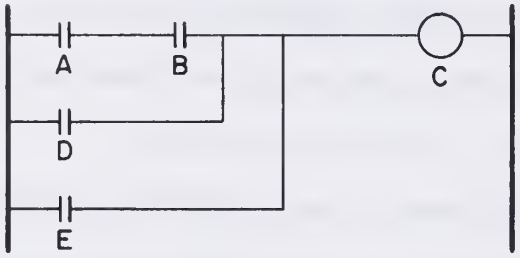
A more subtle error is possible if the RLD used is an exact wiring diagram for the relay logic it describes. For example, consider this innocuous looking RLD.

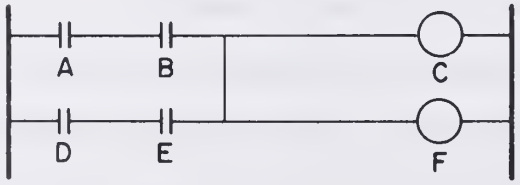


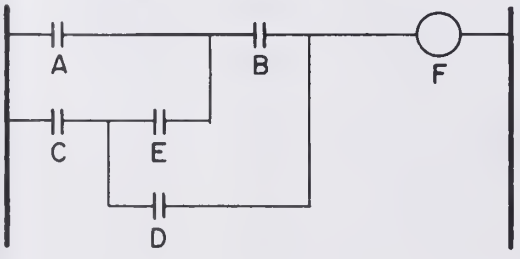
IMPROPER CIRCUIT



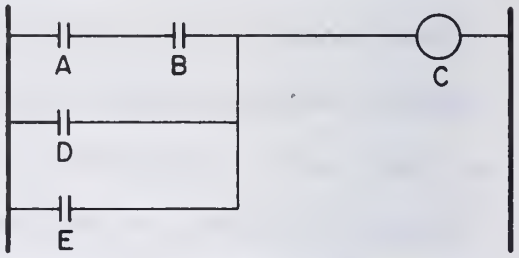


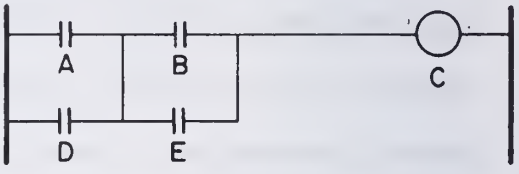


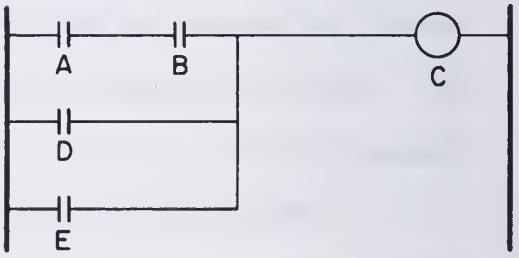


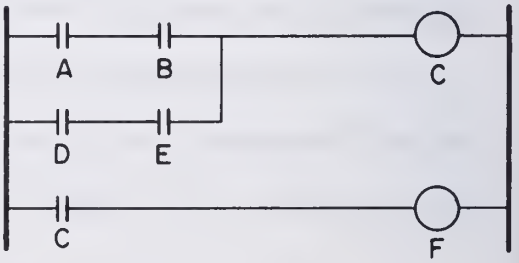


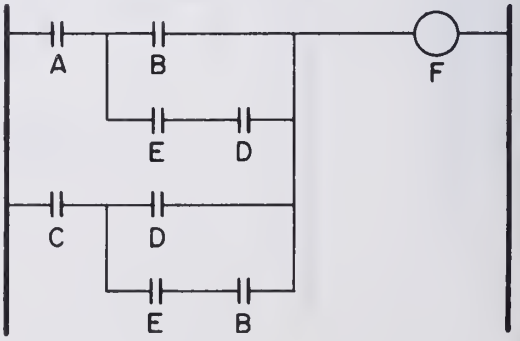
ACCEPTABLE EQUIVALENT











In pure relay logic, conduction from the left-hand power line would proceed bi-directionally along both horizontal and vertical paths. Thus, the Boolean control equation is not

(Equation 1) $X = A*B*C + A*D*E*C + A*D*F$ [* = AND; + = OR]

$= A * (B*C + (D * (E*C + F)))$

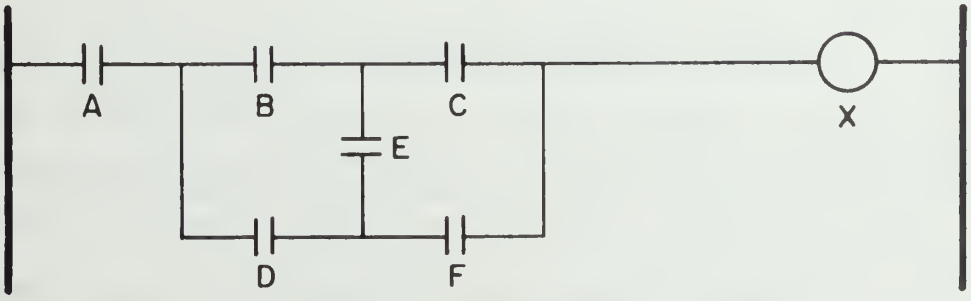
but rather

(Equation 2) $X = A*B*C + A*D*E*C + A*D*F$ [+ A*B*E*F]

$= A * (B * (C + E*F) + D * (E*C + F))$

where the term in brackets represents current flow from right-to-left through relay E.

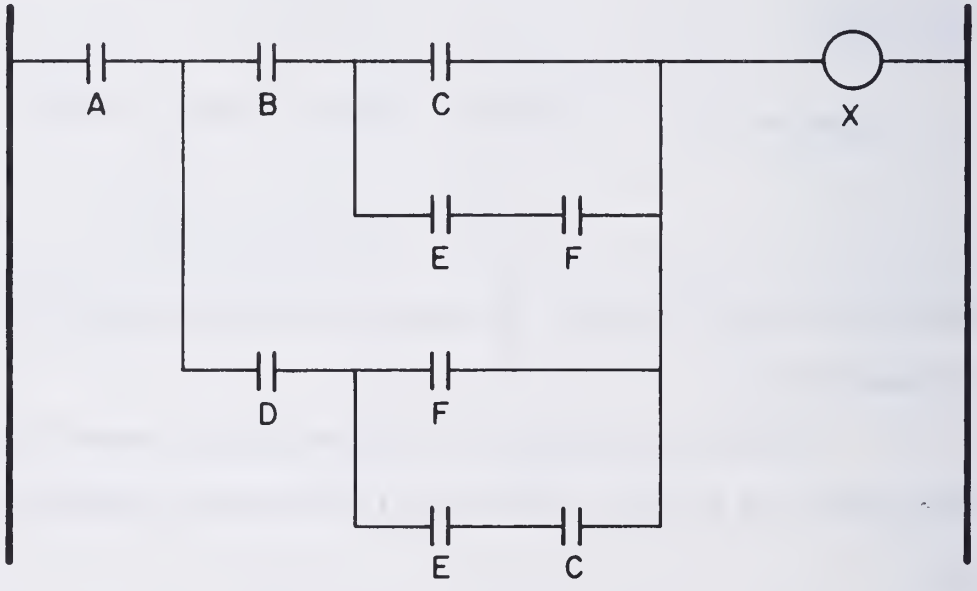
Using the definition of bi-directional current flow on both horizontal and vertical lines makes the above RLD topologically equivalent to



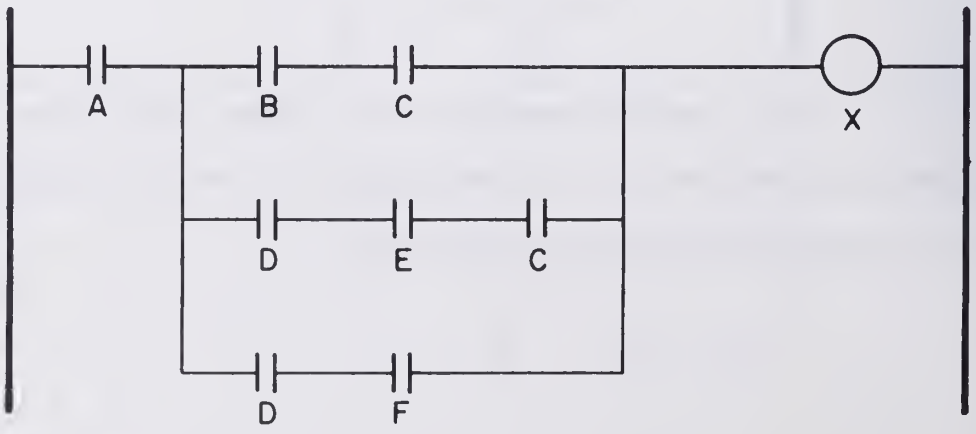
This subtle introduction of a "sneak path" can be a source of great concern, for it is now ambiguous as to which of Equations 1 or 2 truly describes the user's intentions.

The VT-14 solves the ambiguity problem by treating all RLDs containing sneak paths (i.e., those containing relays through which current could flow in both directions) as illegal formats.

Thus, if the original RLD is to be interpreted by Equation 2 where relay E is conducting bi-directionally, the RLD must be redrawn as



However, if the original RLD is to be interpreted by Equation 1 where relay E conducts only left to right, it too must be redrawn to eliminate an illegal format and results in a figure such as:



The implementation of the original RLD, as interpreted by Equation 1 and as drawn above, requires duplicating contacts D and G, which in turn implies more memory space and a longer execution time for the control program.

In summary, VT-14 brings the user closer to the familiar relay symbols than any prior system. Its advantages include a large programming matrix and a conceptually simple pushbutton programming method requiring no knowledge of computer programming languages or Boolean algebra. Some of the system's disadvantages stem directly from poor human engineering, including:

1. Identifying the third relay element (the fifth element positionally) on the first line as (row,column) = (0,4);
2. The necessity of hand calculating the matrix position of an element to be altered, rather than using a movable cursor;
3. Entering (x,y) matrix positions and I/O addresses via thumbwheel switches; and
4. Requiring program entry to be strictly left to right and top to bottom.

Other disadvantages are a direct result of the strict formatting rules and the prohibition against drawing any RLD which, regardless of the user's intentions, could be interpreted as containing a sneak path under the assumption of bi-directional current flow. These restrictions can cause a great deal of redrawing on the part of the user in an effort to satisfy the programming syntax.

2.5 Summary

In the preceding sections we have traced the development of industrial controller software from assembly languages and high-level programming languages to graphically-input relay ladder diagrams. Chapter 3 pursues the development of an even more versatile relay symbol language and the software methods for decoding the resulting RLD.

3. DEFINITION AND TRANSLATION OF THE PROGRAMMING LANGUAGE

3.1 Language Definition

3.1.1 Data Types and Graphic Symbols

Referring to the system software goals as outlined in Section 1.7, we proceed at this point to define a graphic programming language which meets our specifications. First, we must determine what types of variables (graphic symbols) are required by a typical user. Examination of a large number of RLDs produced by industry suggests that the following data types are required.

Input/Output Data Types

1. INPUT--a Boolean value supplied by the real world;
2. OUTPUT--a Boolean value calculated by the control program; and
3. CONTROL--a Boolean value calculated by the control program, useful as an intermediate result or as a monitor checkpoint, but not supplied to the real world.

The programming language should allow the user to specify whether each instance of an INPUT, OUTPUT, or CONTROL variable uses its normal or its complemented value.

Further, it should be possible to latch (retain) the value of OUTPUT and CONTROL varaibles in the event of a power failure. It is neither necessary nor desirable to make all OUTPUT and CONTROL variables

inherently retentive. Suppose an output controlled the status of some machine; in the event of a power failure the controller will stop immediately, but the controlled machine may coast due to momentum. Due to the coasting, the last calculated set of values for OUTPUT and CONTROL variables will no longer represent the true state of the controlled machine. A restart using these incorrect values would be obviously hazardous. For this reason, all nonretentive variables are cleared (reset) upon startup, i.e., all nonretentive outputs are initially off. Only those variables specified as retentive by the user are omitted from this startup initialization. A side effect is that the user's programming logic is simplified by guaranteeing that the controller's outputs will start in a known state.

The D-1001 provides a total of 64 inputs and outputs, 32 in the basic controller and an additional 32 in an optional I/O expander box. A relay or solenoid used in an RLD with an address in the range 1 to 64 will be logically connected to the physical I/O terminal of the same number. Sixty-three control relays are provided and are addressed in the range 101 to 163; these relays have no real world terminals.

Internal Data Types

In addition to 1-bit INPUT, OUTPUT, and CONTROL data, the system should provide digital timers and counters implemented wholly in software. This controller provides eight such timer/counter modules; each module may be user-specified to operate individually as either a timer or a counter. This design permits maximal flexibility by allowing the exact

number of timers and counters, up to a total of eight, to be determined by the specific application rather than by the supplied hardware.

1. TIMER Definition. A timer is a normally-open, timed-closed device which, when activated, times down to zero in 0.1 second increments from an initial preset time which is in the range of 000.0 to 999.9 seconds. The output of the timer is off when the timer is reset and while it is active but not yet timed out (i.e., its current time has not yet reached zero). Control of the timer is accomplished through its two coils, START and HOLD. The value of a timer's current time and preset and the status of its START coil are inherently retentive. The HOLD coil is cleared during system startup.

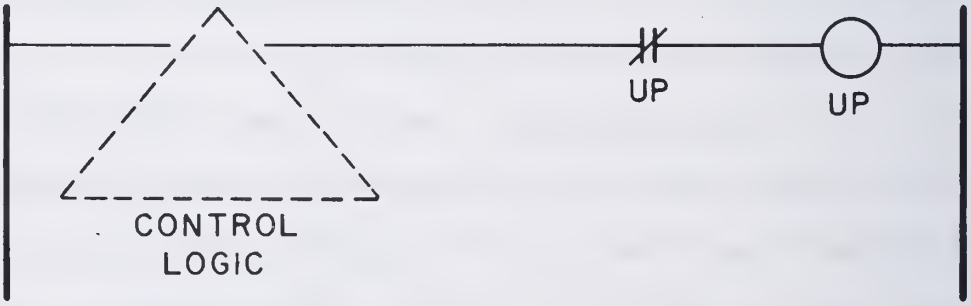
2. Timer START Coil. When the timer's START coil is active (on), the timer continues to time down in 0.1 second increments, provided it has not already timed out; if it has, it will not decrement below zero. When the START coil is inactive (off), the timer is reset (its current time is set equal to its preset time and its output is turned off).

3. Timer HOLD Coil. When a timer is active (START coil on), timing can be temporarily suspended by activating the HOLD coil. If the HOLD coil is off it has no effect.

4. Timer OUTPUT Contact. The timer's OUTPUT is on if the timer has timed out (current time is zero); otherwise, it is off.

5. COUNTER Definition. A counter is a normally-open device which closes when its internal count equals or exceeds a specified setpoint. Its setpoint is a 4-digit number in the range 0000 to 9999. A

counter will neither count up past 9999 nor count down below zero. Control of the counter is accomplished through its three coils: count UP, count DOWN, and count RESET. Each of the three control coils is software-simulated to be leading-edge-triggered. Thus, to count up twice, the count UP coil must be initially off, turn on, turn off, and turn on again. Since a counter is used to record discrete events, independent of the time frame in which they occur, the software-edge-triggered feature eliminates the need for modifying the RLD by adding a blocking contact as shown below.



The value of a counter's current count and setpoint are inherently retentive; its UP, DOWN, and RESET coils are cleared during system startup.

6. Count UP Coil. On its transition from off to on, the current count is incremented by one if it has not already reached 9999. If the current count equals or exceeds the setpoint the output is turned on, else it is turned off.

7. Count DOWN Coil. On its transition from off to on, the current count is decremented by one if it has not already reached zero. If the current count equals or exceeds the setpoint the output is turned on, else it is turned off.

8. Count RESET Coil. On its transition from off to on, the current count is set equal to zero. The output is turned off.

The RESET coil has precedence over both UP and DOWN coils (a counter which is being reset will neither count up nor down). The count UP and count DOWN coils are independent so that a counter may be incremented and decremented in the same cycle.

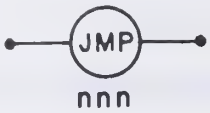
Additional Internal Variables

1. FIRST SCAN. While the OUTPUT and CONTROL data types allow the user to specify whether or not they are retentive, other data types are defined to be inherently retentive. The FIRST SCAN coil is on during the first solution (the first memory scan) of the control program and off for all subsequent solutions (FIRST SCAN is also on during the first program solution after a system restart). This coil can be used directly in the RLD to clear variables which should not be retentive or to activate logic which should only occur upon startup or restart.

2. One Second Clock. The internal time base is 0.1 seconds. From this base a one second clock output (on 0.5 seconds, off 0.5 seconds) is developed which is useful for extending the timing range beyond the normal 999.9 second limit.

As a convenience to the user, two additional nonrelay functions are also provided, JUMP and RESET.

3. JUMP Coil. JUMP is a special function (coil) which, when activated, jumps (does not evaluate) the next sequential nnn pages of



the RLD program. The JUMP feature allows the user to optimize his program, with a significant saving in execution time, by conditionally jumping over segments of his RLD program, where the jump or no-jump decision is specified in purely relay logic. There may be any number of JUMPs per program.

4. RESET Coil. RESET is a special function (coil) which, when activated, resets (clears) the outputs (in column eight) of the next sequential nnn pages of the RLD program. This "master reset" allows the user to optimize his program by using one simple page of relay logic feeding the RESET coil to conditionally clear any number of real outputs. Both memory utilization and execution time are improved when the "master reset" is used, rather than including the reset condition in the definition of each output.

A set of relay symbols are defined in Figure 8 which are useful for representing all of the above data types. In cases where a single symbol may represent one of many variables of the same type, the I/O address supplied by the user resolves any ambiguity.

3.1.2 Programming with the CRT

The program is to be entered via pushbuttons and displayed on a CRT. The individual "pages" of an RLD are defined to be a programming matrix of four rows and eight columns. Each matrix position in the first seven columns may contain a space, horizontal connection, normally open

GRAPHIC SYMBOL	MEANING
	blank
	horizontal connection
	normally open relay
	normally closed relay
	vertical connection
	normally open output
	normally closed output
L	latched (retentive) output
L	latched (retentive), complemented output
T	timer coil
T	complemented timer coil
C	counter coil
C	complemented counter coil
JMP	JUMP coil
RST	RESET coil

Figure 8. Programming Symbols for Drawing Relay Ladder Diagrams on the D-1001

relay, or normally closed relay. A vertical connection to the line below may optionally be added to any symbol on line one, two, or three. The relay element addresses may refer to any input, output, control variable, latch, timer coil, counter coil, or any other internal variable; thus, every variable in the system may potentially participate in the control of any output. Column eight is reserved for outputs (normal or complemented, retentive or nonretentive, timer/counter coils, etc.). Any page may contain one to four outputs, one per line. There are no formatting restrictions with regard to the interconnections on a single page. Figure 9 shows a maximal RLD page in which every possible matrix position and interconnection are in use.

The program is entered interactively using a blinking screen cursor to identify the matrix position currently being programmed or edited. To insert a symbol in any location, the cursor is simply moved into the desired position by depressing the cursor control pushbuttons:

move up one line

move down one line

move left one element

move right one element

Having selected a matrix position with the cursor, any of the relay symbols may now be inserted. The I/O address of the element is entered on a keypad containing the digits 0 to 9. Each address digit entered shifts the current 3-digit element address field (initially 000) one digit to the left and inserts the just-depressed digit in the least-significant

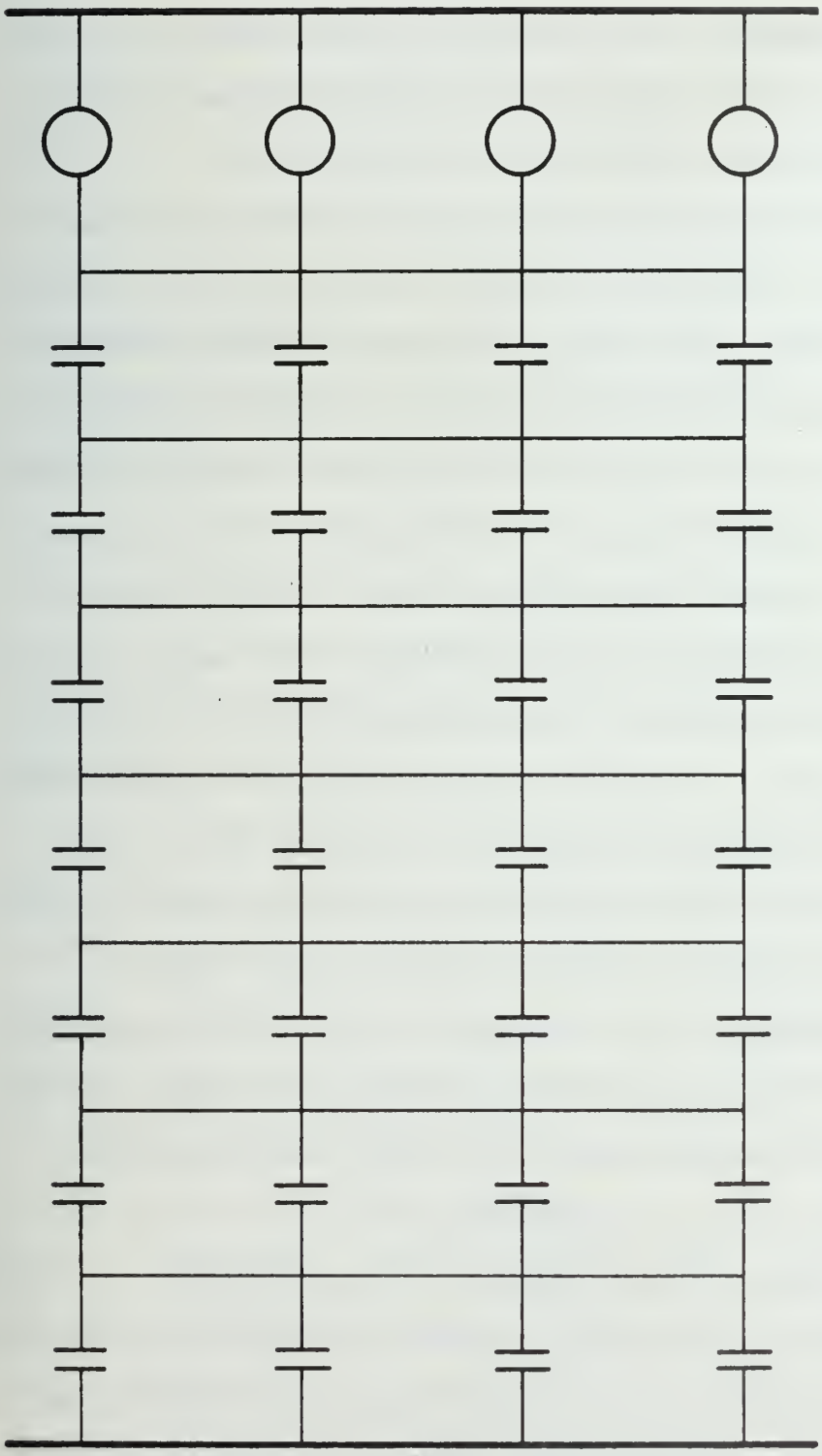


Figure 9. Maximum RLD Page

digit position. Depressing any of the output symbols automatically advances the cursor to column eight, filling intervening blank elements with horizontal connections, and inserts the output symbol.

If an error is made while programming a given page, the user need only reposition the cursor to the matrix position in error and press the symbol and/or address keys corresponding to the desired correction. In this manner the user may program any matrix position, in any order, as well as edit any number of positions in any order. Any symbol may be added, deleted, or altered with only a few keypresses. After the page has been visually inspected, depressing NEXT PAGE in program mode saves the current page and displays a blank next page; in edit mode it replaces the saved page with the page just edited and then displays the next page as recalled from memory. For ease of editing, accuracy of documentation, and enhancement of visual verification of correctness, the translation technique preserves the exact topology of the input RLD so that it can be reconstructed exactly from the internal code at any later time.

When programming a timer or counter, the user may choose to insert his 4-digit presets/setpoints directly through the keypad. All presets/setpoints may be changed dynamically at any time during program execution for system tuning or correction.

When all the pages of a program have been completed, the COMPILE pushbutton is depressed. The graphic translator software package is then invoked which translates the various graphic pages into internal code.

The utility of the programming format is illustrated by the simplicity of its rules:

1. There are four rows and eight columns per RLD page;
2. Column eight is reserved for outputs;
3. Verticals may not extend below line four; and
4. Lines which cross are assumed to connect.

The user should appreciate the fact that it is impossible to introduce a permanent syntax error. Not only is the language designed for freedom of expression, but the only possible syntax errors are detected by the graphic translator:

1. Nonoutput type symbol in column eight;
2. Vertical entered on line four; and
3. Invalid I/O address for relay element.

In each case the invalid keypress is immediately rejected and an audible beep (error signal) emitted.

The switch from program to edit mode is painless and immediate. Since the system decompiles compiled code exactly, the user experiences no disturbing format change when he edits a program. To further aid his editing, a SEARCH function is provided in which a relay symbol and address are given. Each sequential depression of the SEARCH key locates and displays the next page of RLD code in which the given symbol is used in conjunction with the given address.

Other modes, including MONITOR, FORCE, and MEMORY COPY, allow the user to interactively debug a running program by watching relays conduct and outputs close in real time, or by forcing any input on or off. These modes are more fully described in Chapter 5. Once a complete RLD

has been entered, it is compiled and stored in a 1Kx8 RAM within the program loader. After further execution, testing, and possible alteration in RAM, the program may be permanently stored by inserting a 1Kx8 PROM and depressing a "copy RAM" key; the PROM will be automatically programmed to duplicate the content of the RAM. Likewise, a previously compiled program stored in PROM may be transferred to RAM by depressing a "copy PROM" key; the program will be decompiled and made ready for display, monitoring, and/or editing. Programs in RAM and PROM may be verified to be identical by use of the VERIFY key; a CRT message reports the address and content of any mismatch.

In summary, the graphic programming "language" described provides numerous benefits over any other programming system currently available. Its use of standard relay symbols, keypress programming, easy program entry and editing, and reliance on visual verification of correctness allow the user to capitalize on his intuition; drawing his RLD on the CRT is no more trouble than drawing it on paper.

From the user's point of view, the most important advantages accrue from the unrestricted format of the RLD input (unique to the industry), the ability to recreate the exact RLD used as input from the internal code alone (also unique), and the freedom to define up to four outputs per relay page (likewise unique).

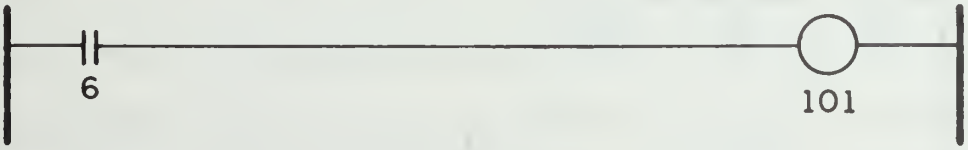
3.2 Language Translation

3.2.1 Definition of an Average Program

Having defined an acceptable graphic programming language, a

significant problem still remains--how to translate the RLD into an internal code in such a way as to extract its Boolean content while preserving its topology. Historically, the imposition of formatting restrictions has served the purpose of reducing the complexity of the translation process. But now we search for a suitable translation scheme which will neither impose format restrictions nor be so complicated as to strain the space and speed limitations of a microprocessor-based system.

Before examining specific algorithms, we should first establish a set of realistic test cases (benchmarks) so that alternative translation schemes can be compared. The benchmark programs should be chosen such that they are neither trivially simple nor improbably complex, but rather approach some kind of "average complexity." By observation of existing control programs, we can determine that an "average" program for our purposes will contain approximately 300 relay elements (inputs, outputs, and control relays) with a density of approximately 10 relay symbols per page. Of course, some pages will be as simple as two elements



while others may approach the complexity of the maximal matrix (as shown in Figure 9) containing all 32 elements and all possible interconnections. Three "typical" pages from the benchmark program are shown in Figure 10. Data collected from translation of the benchmark program by each prospective method can predict the performance of the set of anticipated "average" programs. The most valuable quantitative results are the projected

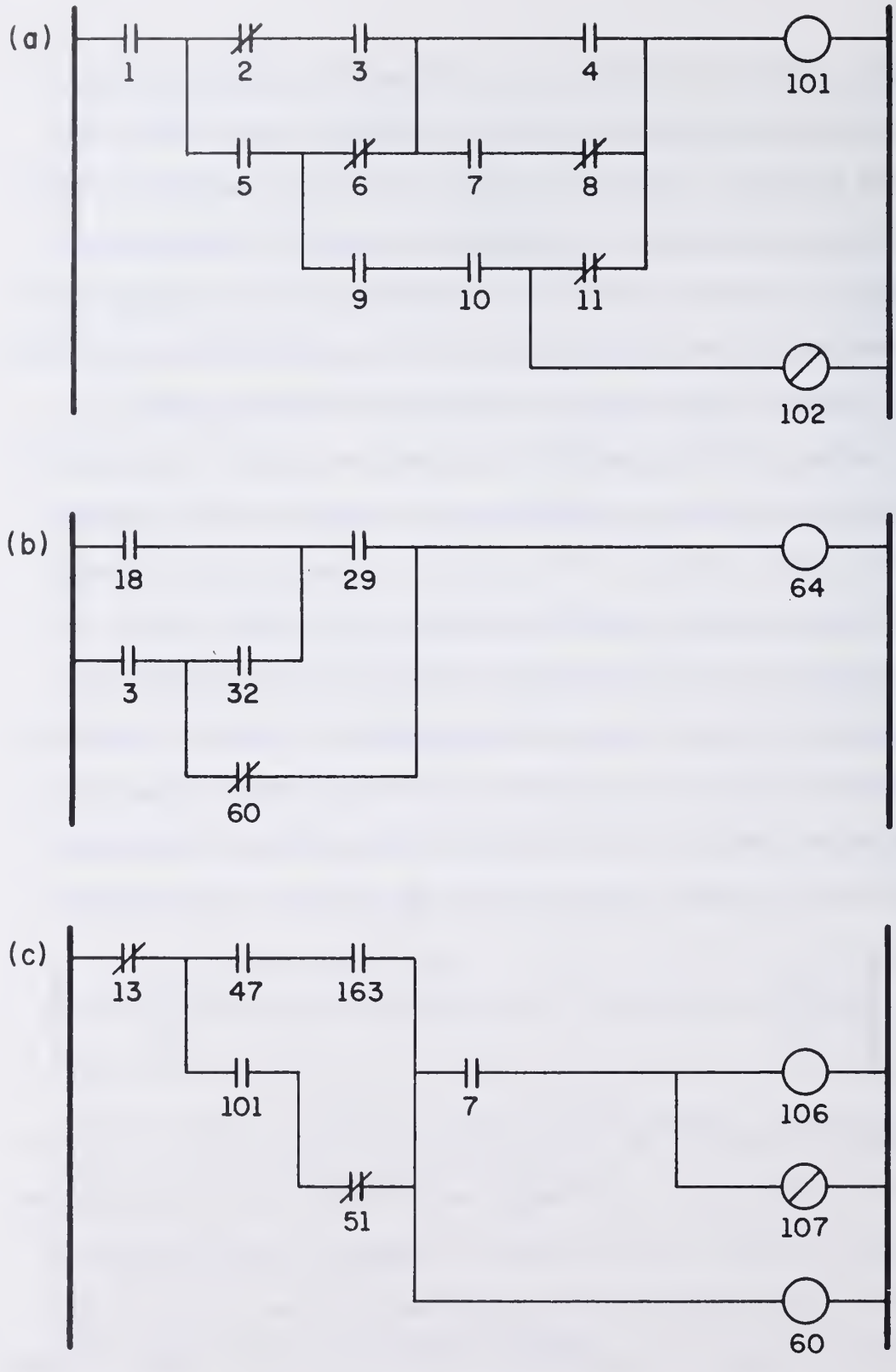


Figure 10. Three Pages from the Benchmark Program

execution response times and program memory requirements; subjectively, one should also consider the complexity of the various translation schemes and the impact of their complexity on such parameters as system memory requirements and translation time.

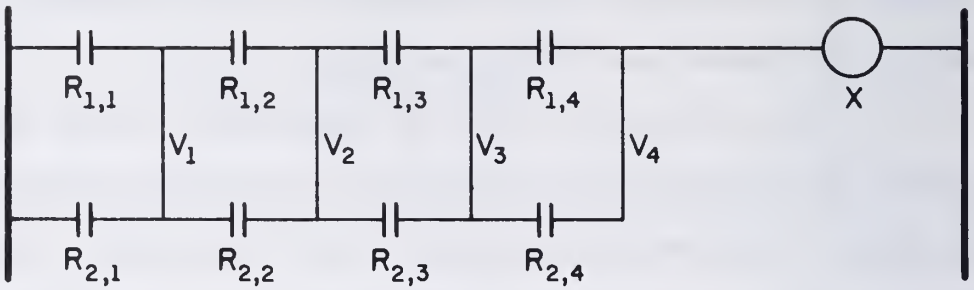
Into what should the RLD be translated? The two basic alternatives are to produce executable code for the chosen microprocessor or to produce an intermediate text which can be interpreted. Although the former offers the advantage of high-speed execution, the latter scheme will conserve memory space as well as simplify conversion to a different microprocessor in some future version of the hardware.

Since the target machine is a microprocessor, let us first consider schemes which produce directly executable microprocessor code.

3.2.2 Translation via Boolean Templates

Given a matrix of size n rows by m columns with n(m-1) possible internal nodes, it is possible to generate a single, generalized Boolean equation which would, for each output in column m, include a term for each possible current path through the matrix as a function of n(m-1) possible relay elements and the (n-1) (m-1) possible vertical connections. Such a Boolean template is shown in Figure 11 for a condensed matrix of 2 rows and 5 columns.

Given the master equation template, each page of the user-generated RLD could be mapped directly onto the master equation by substituting relay addresses for each relay element (along with an indication of whether to use the true or complemented logic value at that



$$
\begin{aligned}
X = {} & R_{1,4} * (R_{1,3} * (R_{1,2} * (R_{1,1} + V_1 * R_{2,1}) \\
& + V_2 * R_{2,2} * (R_{2,1} + V_1 * R_{1,1})) \\
& + V_3 * R_{2,3} * (R_{2,2} * (R_{2,1} + V_1 * R_{1,1}) \\
& + V_2 * R_{1,2} * (R_{1,1} + V_1 * R_{2,1}))) \\
& + V_4 * R_{2,4} * (R_{2,3} * (R_{2,2} * (R_{2,1} + V_1 * R_{1,1}) \\
& + V_2 * R_{2,1} * (R_{1,1} + V_1 * R_{2,1})) \\
& + V_3 * R_{1,3} * (R_{1,2} * (R_{1,1} + V_1 * R_{2,1}) \\
& + V_2 * R_{2,2} * (R_{2,1} + V_1 * R_{1,1})))
\end{aligned}
$$

Figure 11. Boolean Template for a Small Matrix

address) and logic ones for each vertical connection and straight-through horizontal connection. Logic zeroes are substituted for all absent horizontal elements and all vertical connections which are omitted. Compilation would then proceed normally on what has now been reduced to a very standard assignment statement. As can be readily deduced from the example, the number of terms in the Boolean equation grows rapidly with each additional row or column. Likewise, the number of common subexpressions increases so rapidly that, for any reasonable matrix size, an optimizing compiler would be absolutely required if the output code is to be acceptable in terms of quantity or execution time.

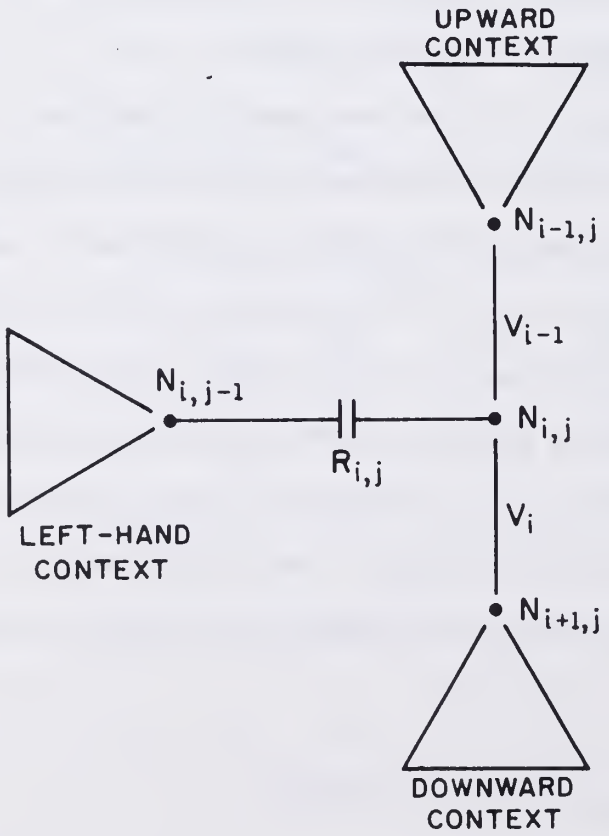
The advantage of the Boolean equation template is that both the compilation and optimization techniques are well known [29,30,31] and could be expected to produce high quality microprocessor code. On the other hand, the number of terms in the equation and the complexity of the optimization pass would require a large amount of microprocessor memory for both the translation program (in ROM) and the working store (in RAM). The time of translation would be extended by the optimization pass, although this is a minor consideration. Most importantly, the optimization of the microprocessor code would prohibit using the code sequence itself to reconstruct the original RLD.

3.2.3 Translation via Recursive Traversal

An alternative to the Boolean equation template is a recursive matrix traversal of each relay page for each output on the page. One could traverse the matrix and build a parse tree, thus achieving a data

structure containing a parse of the equivalent Boolean equation without actually creating or manipulating the equations explicitly.

At each node $N_{i,j}$ of the matrix there exists a relay element $R_{i,j}$ (possible null) between nodes $N_{i,j}$ and $N_{i,j-1}$, a possible vertical extending downward from the line above connecting nodes $N_{i-1,j}$ and $N_{i,j}$, and a possible vertical extending downward to the line below connecting nodes $N_{i,j}$ and $N_{i+1,j}$. A generalized picture shows node $N_{i,j}$ surrounded by its upward, downward, and left-hand context.



Beginning the traversal at an output (column m) and moving into the output's left-hand node $N_{i,m-1}$ from right to left, one begins a

recursive search upward, downward, and leftward to explore all paths which may lead backward (right to left) from the output to the power line. Figure 12 shows a simple RLD and its optimized parse tree produced by recursive traversal. With the parse tree in hand, one may now submit it to a code generator for production of microprocessor code.

Due to the structure of the previous example (the OR branches were uncomplicated) the recursive nature of the search produces no serious side effects. However, when the example is changed to complicate the OR branches, as in Figure 13, the number of paths increases significantly and both traversal time and working storage are significantly increased. Of course, the introduction of multiple paths through the matrix increases the number of common subexpressions (common subtrees) and again requires a code optimizing compiler to remove them. As before, optimization removes redundancy and thereby prohibits an exact reconstruction of the input RLD.

3.2.4 Expected Results of Microprocessor Code

What are the space/speed characteristics of directly executable microprocessor code? Assume that, regardless of the translation method used, either optimal or near-optimal code can be generated for a one-accumulator machine. See Figure 14 for an example. Using the instruction set and speeds for a MOS Technology 6502 microprocessor, the benchmark programs suggest that the "average" 300 relay symbol program would require about 3000 (8-bit) bytes of code and an execution time of 4 ms.

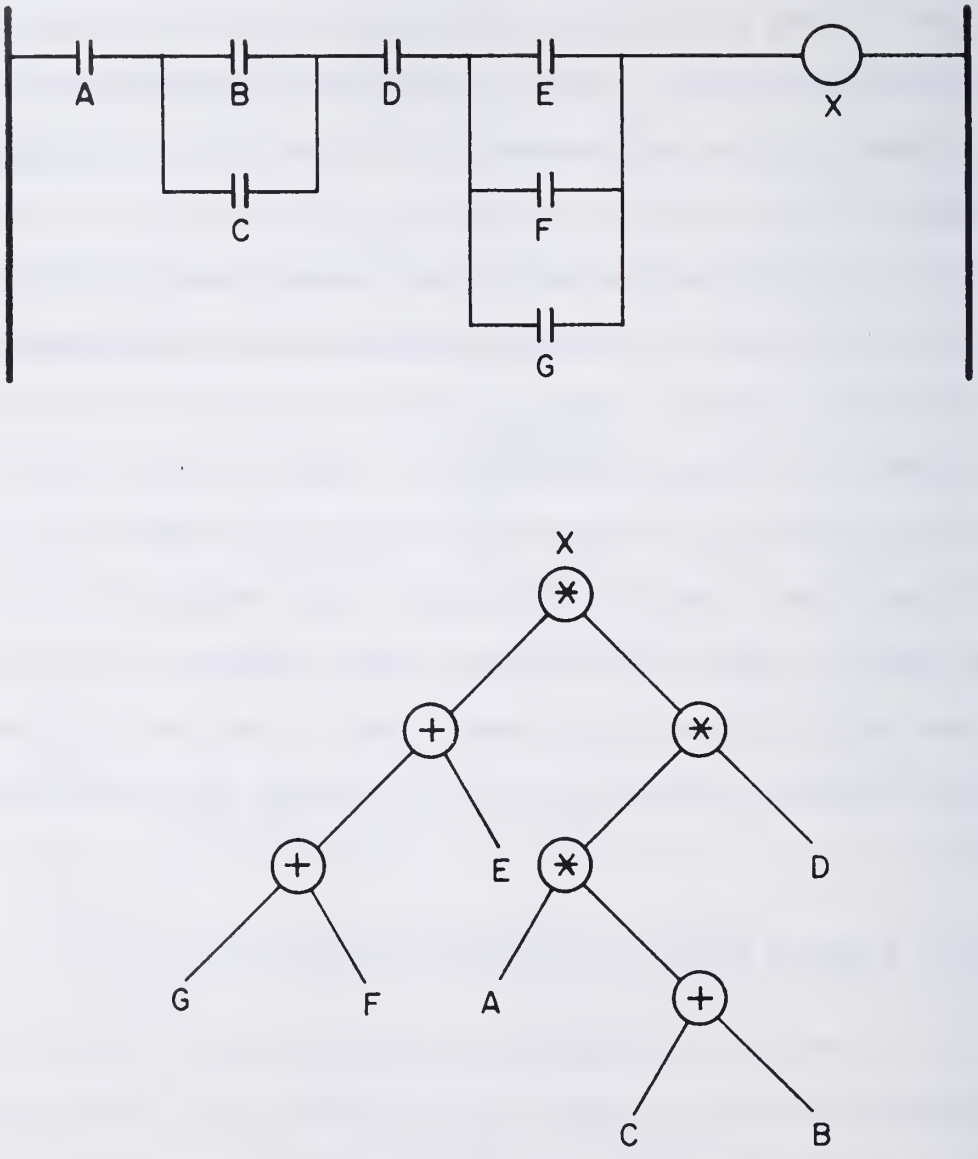


Figure 12. A Simple RLD and Its Optimized Parse Tree

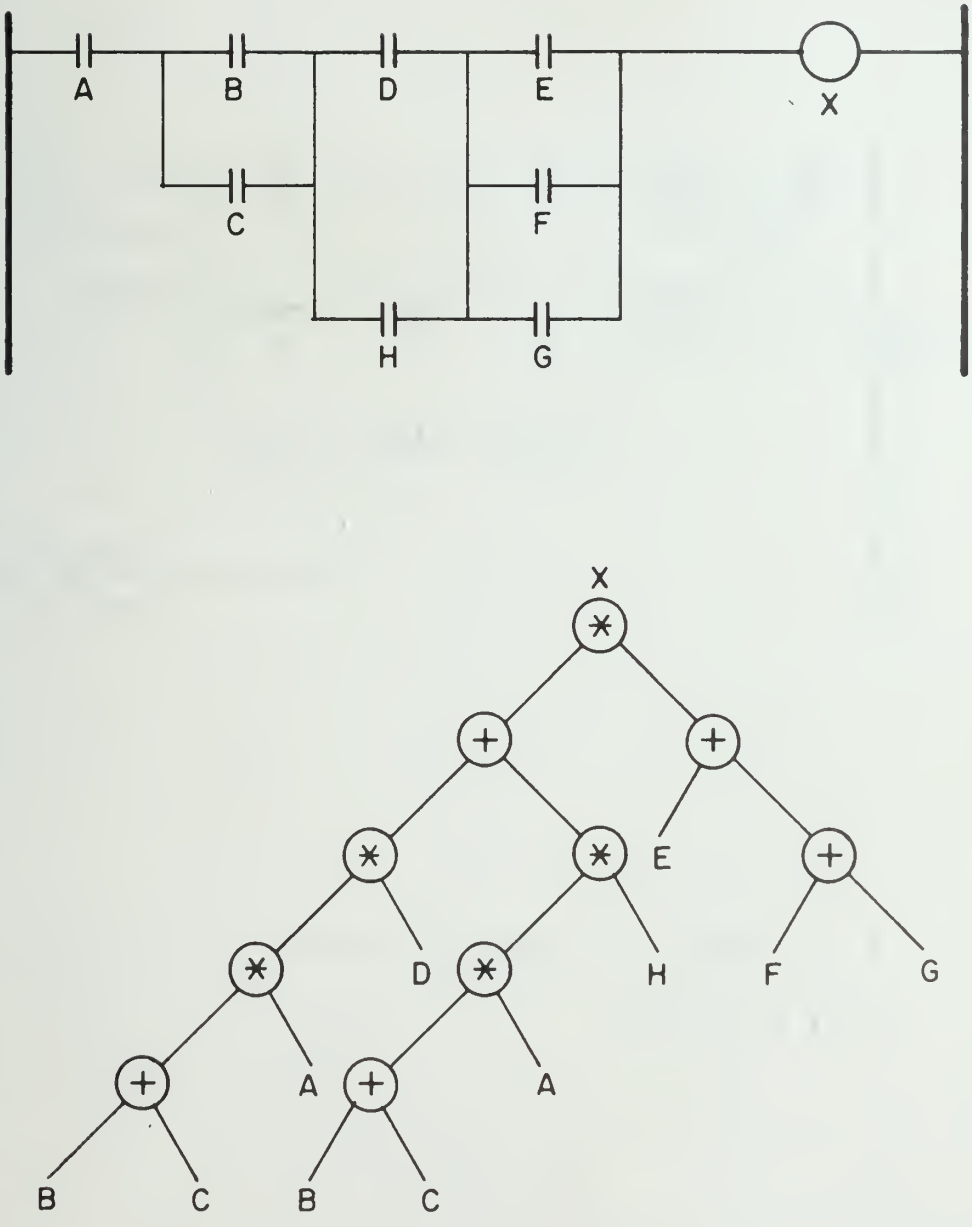
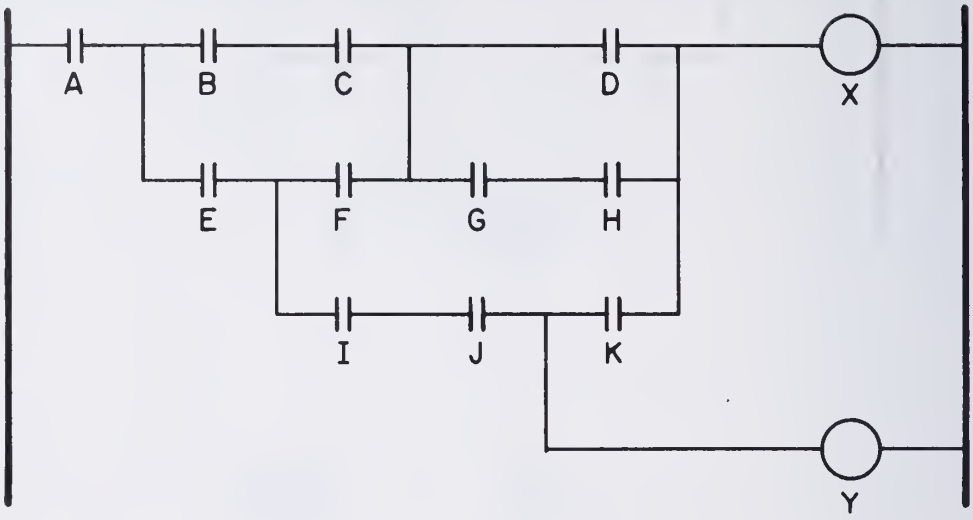


Figure 13. A More Complicated RLD and Its Optimized Parse Tree



$$X = A * ((B * C + E * F) * (D + G * H) + E * I * J * K)$$

$$Y = A * E * I * J$$

Figure 14a. RLD and Equivalent Boolean Equations

```
LDA     B
AND     C
STA     TEMP1
LDA     E
AND     F                       analysis:
ORA     TEMP1                   21 instructions
STA     TEMP2                   57 bytes of code
LDA     G                       78 microseconds execution time
AND     H
ORA     D
AND     TEMP2
STA     TEMP3
LDA     A
AND     E
AND     I
AND     J
STA     Y
AND     K
ORA     TEMP3
AND     A
STA     X
```

Figure 14b. Directly Executable Microprocessor Code

While the execution time is well within the desired range, the memory requirement is not. The information density (memory bytes required per relay symbol) is low due to the bit masking and shifting required by the microprocessor architecture and the overhead of keeping two copies of the output and control variables (necessary to guarantee equation independence, see Chapter 4).

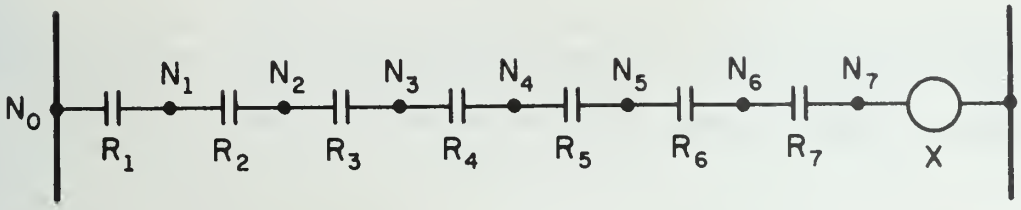
The excess memory requirement is not the only problem, however. Since the code embodies the logic, but not the topology, of the RLD input, the original geometry cannot be recovered from the microprocessor code alone. Additionally, execution-time optimization, such as recognizing that the evaluation of a lengthy expression is unnecessary if the result is to be ANDed with a zero, is possible only at the expense of even more memory. Further, should the system be programmed by hand, rather than by the graphic translator, using microprocessor code as the source language would require the user to be very familiar with a specific microprocessor architecture and its assembly language.

In summary, the main advantage of directly executable microprocessor code is its speed of execution; however, it does not facilitate recovery of the input RLD, execution-time optimization, or hand programming. Therefore we will examine interpretive codes which do.

3.3 Execution-Time Optimization

Directly executable microprocessor code does not permit execution-time optimization without adding the appropriate tests and branches to the output code, thereby increasing the amount of code generated.

Consider this one line RLD page.



Field measurements have shown that any given relay element, considered independently, tends to be nonconducting approximately 60 percent of the time. Thus, statistically, the probability of there being a conductive path from node N_0 to node N_7 depends upon the number of series relays between them. For the figure above, the probability of the output X being active is:

Number of Relays in Series	Probability of X being Active
1	0.400
2	0.160
3	0.064
4	0.026
5	0.010
6	0.004
7	0.002

For any one column of four elements in parallel, the probability of column conduction equals the probability of one or more elements conducting, which equals one minus the probability of all four elements being off.

$$1 - (0.600)^4 = 1 - 0.130 = 0.870$$

For the maximal matrix (Figure 9) the probability of conduction from column to column depends upon the number of columns in series:

Number of Four-Element-in-Parallel Columns in Series	Probability of Column Conduction
1	0.870
2	0.757
3	0.659
4	0.573
5	0.498
6	0.434
7	0.377

From this elementary analysis we see that the probability of an output's being on is bounded between 87.0 percent (one four-element-in-parallel column feeding one output) and 0.2 percent (seven single elements in series feeding one output). Thus there exists a moderate degree of optimization to be gained by recognizing that once the left-to-right flow of current has stopped at nodes $N_{i,j}$, $j = 1,2,3,4$, then all nodes $N_{k,j}$, $j = 1,2,3,4$ and $k \geq i$ are also logic zeroes. Since the outputs in column eight can be complemented we cannot discontinue execution of the page, but if evaluation proceeds left to right, column by column, we could branch forward to the code for column eight and resume execution there.

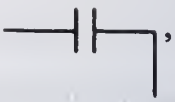
This option of checking for column conduction provides a reasonable compromise between (1) introducing massive amounts of overhead (code and time) to check conduction at the element level and (2) introducing no overhead but wasting execution time by blindly solving a "dead" page.

The desirability of high information density, simple recovery of the input, and low overhead execution-time optimization led to the development of column-oriented macrocodes which achieve all three objectives.

3.4 Interpretive Internal Codes

An interpretive code can be used to increase the information density as well as to encode the geometry of the input RLD at the expense of a larger software system (an interpreter) and additional execution time. Still, such a tradeoff would be economically favorable if it reduced the investment in PROM chips required for the translated RLD program by more than it increased the investment in ROM chips required for the whole operating system (assuming execution time remained satisfactory).

Such considerations led to the development of a series of interpretable macrocode instruction sets. One such instruction set encoded each graphic symbol in an 18-bit opcode plus addresses format such as

where	symbol	identifies an element, such as  ,
	x	is its row number,
	y	is its column number, and
	I/O address	is the address of the I/O terminal to which the element was physically attached.

The primary advantage was that the RLD reconstruction was easily accomplished since the element's matrix position was carried explicitly within the code. However, due to the nature of microprocessors, the decoding of the instructions (involving extensive bit masking and shifting) accounted for much more execution time than did the implementation of the encoded RLD logic. Further, execution-time optimization was not really feasible unless the code order was restricted to be column oriented, which defeated some of the generality of the (x,y) matrix address encoding. Also, the 18-bit memory was not compatible with most microprocessors.

Another implementation based on the same scheme dropped the (x,y) matrix position from the encoded instruction and instead made the codes position dependent (top to bottom within columns and columns ordered left to right). This scheme permitted a 300 relay symbol program, plus significant spaces and horizontal connections, to be encoded in 800 8-bit bytes, but with an execution time of nearly 40 ms. This situation was the opposite of that encountered for directly executable microprocessor code, for now the memory requirement was satisfactory but the execution response time was too long.

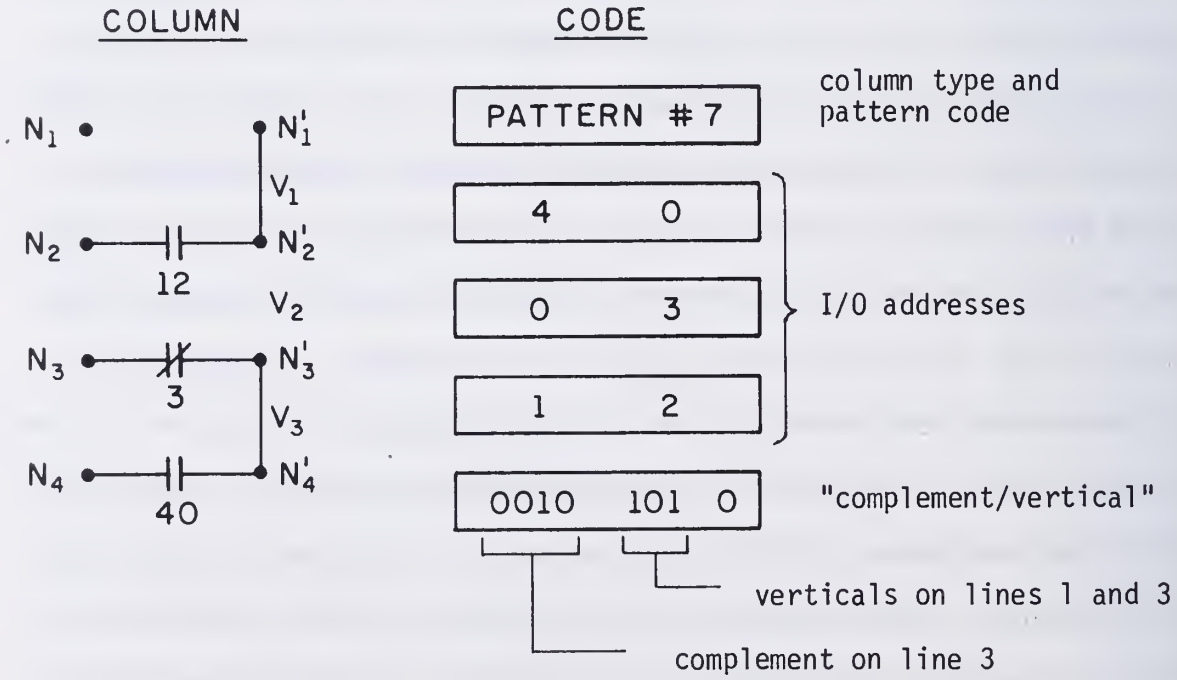
As a result, yet another type of macrocode was sought which would require less overhead for decoding and encourage execution-time optimization.

3.5 A Column-Oriented Code

The solution to these problems lies in developing a macrocode which is simple to define, easy to decode, and which arises naturally from the input RLD. It should be noted that in an RLD depicting true relay logic, conduction would be bi-directional on both horizontal and vertical lines. This once again introduces the problem of sneak paths as discussed in Section 2.4. Control engineers first handled the problem by agreeing to eliminate relays from the vertical lines, thus making sneak paths easier to spot, but they were delighted to find that programmable controllers could simulate one-directional conduction on the horizontal lines, thus eliminating sneak paths altogether. To be exact, the new controllers do not implement relays, but rather relays with diodes. Since the code is interpretive, we can define conduction any way we wish; the D-1001 implements left-to-right conduction on horizontal lines and bi-directional conduction on the vertical lines. Column-oriented code has the advantage of simplifying implementation of left-to-right conduction without sacrificing any execution-time optimization.

To define an adequate column-oriented code, note that in any one of seven relay columns on a page there can be only 16 positional combinations of relays in any one column; therefore, we define 16 distinct "relay column patterns." The pattern number (0 to 15) identifies the

goemetry of the column (pattern # 0 indicates no relays, pattern # 1 indicates one relay on line 4, . . ., pattern # 15 indicates one relay on each of lines one, two, three, and four). Additional memory bytes identify the particular I/O address of each relay element in the column. One additional byte indicates which, if any, of the relay elements are complemented (normally closed) and which, if any, of the three possible vertical connections between lines are present. An example follows.



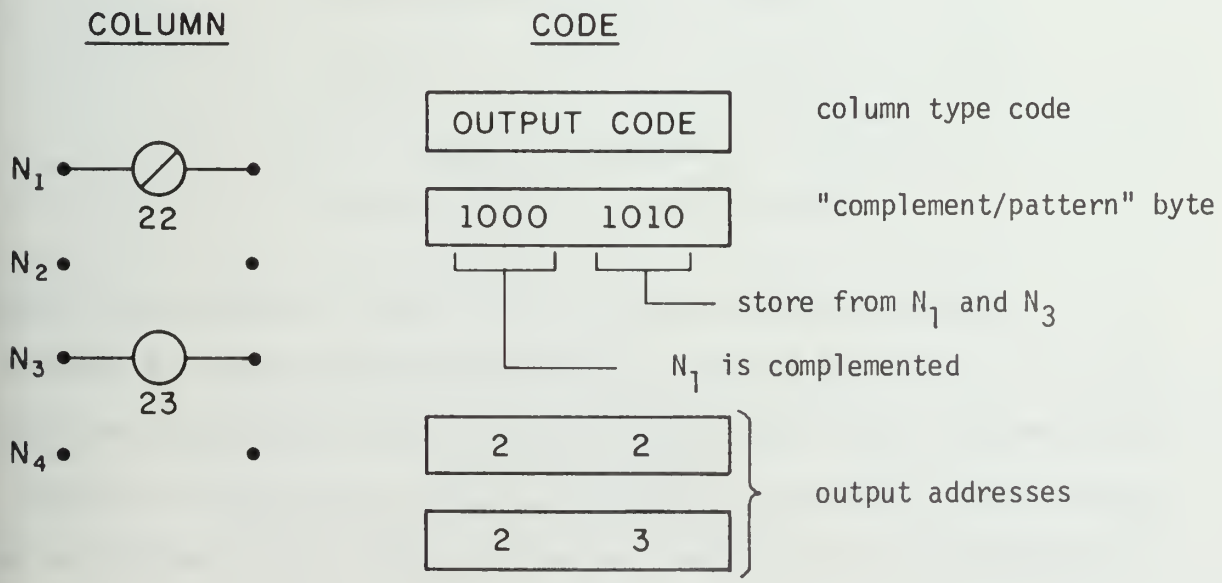
"Executing" the j^{th} column now reduces to fetching the dynamic status of the relay elements of the column, $R_{i,j}$, i = 1,2,3,4, ANDing this 4-bit vector with the left-hand nodes $N_{i,j-1}$, i = 1,2,3,4, and creating a new set of right-hand nodes $N'_{i,j}$, i = 1,2,3,4, after accounting for the influence of complemented variables and additional propagation due to vertical connections.

Since the "solution" of the column conduction problem is a function of fifteen variables,

$$N'_{i,j} = f(N_{i,j-1}, R_{i,j}, C_i, V_k) \quad i = 1,2,3,4, \quad k = 1,2,3,$$

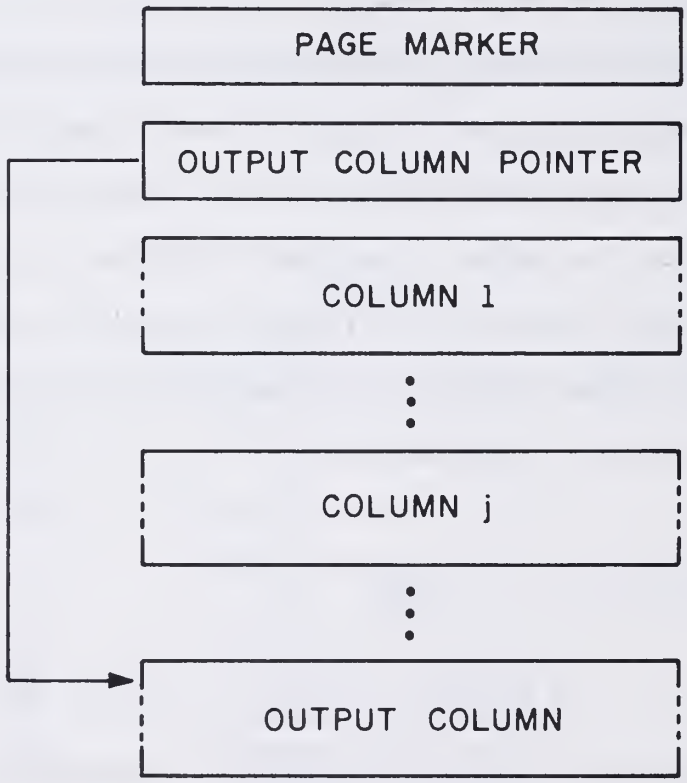
it is speedily solved by table lookup.

Likewise, the output column (column eight) could have been segmented into different patterns for each different type of output column possible, but the large number of possibilities makes this approach impractical. Rather, the output column code identifies the code segment as an output column, identifies the lines from which an output is stored, indicates if it is complemented and to what output address the computed result is to be stored (see below).



A similar approach serves for the JUMP and RESET columns.

The code segments for each page are assembled in column order, left to right, and surrounded by a page marker and a pointer to the output column code as shown below.



Whenever execution detects that a newly calculated set of nodes $N'_{i,j}$, i = 1,2,3,4 are all zero, the internal program counter is advanced to the output column code, thereby omitting both interpretation and execution of the intervening column code. Finally, the code for all pages is assembled along with prelude and postlude blocks which simplify system startup and optimization.

The advantages of the scheme are numerous, including:

1. Simplicity--simple enough to hand code if necessary;

2. Fast--worst case (no optimization branches) for the 300 symbol benchmark program is 25 ms;
3. Economical--a 1Kx8 program PROM (1 chip) will hold a 600 symbol RLD;
4. Input recovery--the exact topology of the input RLD is wholly specified within and recoverable from the internal code; and
5. Moderate software complexity--the operating system for the controller (including the interpreter) is 2K bytes; the operating system, interface, and graphic translator/editor/ monitor for the program loader is 8K bytes.

4. THE DIRECTOR 1001 CONTROLLER

4.1 Hardware

4.1.1 The Microprocessor

Having determined that a microprocessor was a better choice hardware than either a minicomputer or a custom-designed, discrete-component CPU, there remained the task of choosing one from among the thirty different models available. The critical factors influencing the choice included computational speed, compatibility with the memories, versatility and power of the instruction set, word length, ease of interfacing the CPU to the rest of the system, and cost.

Speed

The instruction execution times of currently available microprocessors range from the very slow (about 20 μs for an Intel 8008) to the very fast (about 300 ns per microinstruction for the Intel 3000). Of course, comparing just instruction execution times alone is pointless; a somewhat slower processor with more powerful instructions may well be a better choice in terms of both program execution speed and the amount of memory required for the operating system. Initial tests, benchmarks, and simulation showed that neither end of the speed spectrum was well suited for use in an industrial controller. While the Intel 8008 could not come close to running the benchmark program within its design goal of 20 ms for a 300 symbol program, the Intel 3000 was excessively fast and

its speed alone was not enough of an advantage to offset the additional complexity of a microprogrammable, bit-sliced microprocessor whose Schottky bipolar technology suggested poor tolerance of electrical noise. Additional coding and testing indicated that a more conventional fixed-instruction-set, fixed-word-length, intermediate speed (1 to 5 μs per instruction) processor would be acceptable.

Compatibility

Having a high speed microprocessor implies having high speed memories if the speed of the microprocessor is not to be wasted by waiting for the memories. The design goals of having an operating system in ROM, a user program in PROM, and system I/O and data areas in RAM again argued in favor of an intermediate speed processor/memory system since the price of high speed memories increases much faster than their cycle time decreases. Another hardware consideration is that of matching the voltage and current requirements of the memories with those of the microprocessor to avoid an excessively complex and expensive power supply.

Instruction Set

Of prime importance to the operating system implementer is the power of the processor's instruction set. Sample programs designed for measuring system response time showed that since the operating system must interpret the user program, an instruction set rich in various addressing modes (such as the Motorola 6800 [33]) was ultimately more of a speed advantage than a processor which had a faster basic instruction set, but

which required more instructions to implement the same functions (Intel 8080 [34]). Since the operating system makes extensive use of its own system data areas, it was also judged advantageous to have a processor which could use faster-than-average instructions when working wholly within the system data area (like the "page zero" instructions of the Motorola 6800 and MOS Technology 650x [35,36]).

Word Length

The microprocessor is forced to handle a number of data types, including 1-bit Boolean status conditions, 4-bit BCD digits, 8-bit system variables, a 10-bit internal program counter for the 1K user program, and both 8-bit and 16-bit I/O addresses. The 4-bit CPUs (e.g., Intel 4004 [34]) proved to be too slow when attempting to perform arithmetic or data movement; the 16-bit CPUs (e.g., IMP-16 [37] and LSI-11 [38]) could not justify their additional cost in terms of their speed advantage when performing only occasional arithmetic. The 8-bit standard CPUs which offer both binary and BCD arithmetic modes were found to be an excellent compromise. The bit-sliced models offered no particular advantage since the CPU length was almost certain to be 8 or 16 bits to accommodate BCD input and output.

Interfacing

The Motorola 6800 and MOS Technology 650x treat all peripherals as memory and have no special I/O instructions. As a result, interfacing an external device requires only the decoding of a 16-bit address and

the manipulation of timing and synchronization signals. Additionally, having no specific I/O instructions both simplifies the coding of the operating system and increases execution speed, since any wait time for a slower device is automatically accounted for by the hardware READY line, rather than by software semaphores.

Cost

The LSI-11 would have provided an extremely powerful CPU had it not been for the fact that its cost alone almost exceeded the expected selling price of the D-1001 controller. Based on the five criteria above, the choice eventually narrowed to three possible units: Intel 8080, Motorola 6800, and MOS Technology 6502. At the time when the CPU decision had to be made the cost in unit quantities for the microprocessors alone was $69, $48, and $25, respectively (of course, prices have since changed). Also, the 6800 required an external clock which would have boosted its price another $10. In terms of cost alone, the advantage was clearly with the 6502.

In view of these considerations, the ultimate choice was the MOS Technology 6502 [35,36] which provides a fixed instruction set of moderate speed (3 to 7 μs), is rich in addressing modes (accumulator, immediate, page zero absolute, page zero indexed, absolute, absolute indexed, implied, relative, indexed indirect, indirect indexed, absolute indirect, and stack), is capable of faster execution in the system data area by 15 to 25 percent, allows 8-bit arithmetic in either binary or BCD arithmetic modes, has no special problems interfacing to external devices, and provides a definite cost advantage. The most significant

disadvantages are its single accumulator (vs. two in the 6800), its two 8-bit index registers (vs. two 16-bit index registers in the 6800 and seven 8-bit general purpose registers in the 8080), and its inability to use the two index registers interchangeably in some addressing modes.

4.1.2 The Memories

The microprocessor must deal with six different memories: the operating system in ROM, the user program in PROM, the system data area in RAM, an I/O buffer area in a RAM which is battery powered to guard against data loss due to power failure, a communications RAM which shares information between the controller and program loader, and an additional user program in RAM used during system startup and debugging.

Operating System (PROM/ROM)

Designed and coded to fit two 1K x 8 ROM chips, the operating system contains all the software modules to initialize the system data areas during startup, perform input from and output to the real world terminals, interpretively execute the user program, handle periodic arithmetic for timers and counters, respond to interrupts from the timer clock and the program loader, and communicate asynchronously with the program loader. The operating system occupies a 2K address space from $F800 to $FFFF ("$" indicates a hexadecimal address). The initial versions of the operating system are programmed in Intel 2708 1K x 8 PROMs, while the final version is destined for ROM.

User Program (PROM)

When the user has determined that his program is correct, he may instruct the program loader to produce a PROM version of his compiled program. This Intel 2708 1K x 8 PROM is then physically transferred from the loader's front panel (see Chapter 5 for front panel description) to the controller itself. The transfer of the PROM chip requires the removal of the controller's front panel (providing protection from heat, humidity, electrical noise, and hostile atmosphere) and inserting the chip into the labeled socket. The PROM's 1K address space is located at $2000 to $23FF, with a contiguous 1K ($2400 to $27FF) available in an optional expander box for use with extraordinarily long programs.

System Data Area (RAM)

The operating system needs its own data area for its own internal variables involved in program interpretation and communication with the loader. Since the majority of this information involves 8-bit variables, a 128 x 8 NMOS RAM is used. To gain a speed advantage when working in the system data area, the chip is positioned in the upper half of the page zero address space ($80 to $FF). The processor's stack is set via hardware and software to occupy the highest addresses of this area, growing downward from location $FF.

Input/Output Buffer Area (Retentive RAM)

The I/O buffer area serves two purposes. As an input snapshot area, it records the current value of every input variable when OS scans

inputs between each program iteration. This feature guarantees that an asynchronous change in an input will not result in a solution based on two different values for the same variable. Likewise, as an output buffer area, it permits all outputs to the real world to change at the same time, thereby reducing the probability of race conditions and delayed feedback effects among outputs.

Secondly, OS invests additional code and memory space to keep two copies of every output and control variable--the value just computed by the most recent program solution and the value currently being computed by the program iteration. The purpose of this extra overhead is to guarantee position independence of the pages of the relay ladder program (i.e., any one program, viewed externally, appears to execute identically regardless of how its constituent pages are permuted). "Position independence" is of significant value for three reasons: (1) as with inputs, it guarantees that no one program iteration uses a multivalued output or control variable; (2) it simplifies programming by removing any execution-time influences attributable to the order of programming; and (3) it simplifies editing since additional pages may be inserted anywhere among existing program pages without causing obscure side effects throughout the remainder of the program.

The I/O buffer is a 256 x 4 CMOS RAM with battery backup; in case of power failure its content is retentive so that the state of the controlled machine (inputs, outputs, control relays, timer/counter counts, presets, and coils) is not lost. As explained in Section 3.1.1, those

variables which should not be inherently retentive are reset under program control during system startup.

Communications RAM

This 512 x 8 RAM provides storage for two-way communication and data sharing between the controller and the program loader. Under control of the loader this RAM is switched into and out of the controller's address space at locations $2E00 to $2FFF. It is fully described in Chapter 6.

User Program (RAM)

This 1K x 8 (optionally 2K x 8) RAM contains the user program while it is being debugged. Like the communications RAM, it is switched into and out of the controller's address space (locations $3000 to $33FF) under control of the program loader. It is fully described in Chapter 6.

4.1.3 Peripherals

4.1.3.1 Input/Output Modules

The hardware necessary to convert input voltages into TTL logic levels is provided on plug-in PC boards with four separate inputs per board. Similarly, plug-in output boards convert TTL logic level outputs, four per board, into voltage and current levels sufficient to drive moderate loads directly. Light emitting diodes (LEDs) indicate the status of each output. The basic capacity of the D-1001 is 32 inputs and/or outputs in any combination, expandable to 64 I/O.

4.1.3.2 Timer Clock

Independent of the 1 MHz clock driving the microprocessor, the D-1001 contains a crystal-controlled clock generating interrupts on the IRQ line every 0.1 seconds. If interrupts are enabled (under software control), the operating system uses the clock interrupt as a time base for updating the software timers.

4.1.3.3 External Communications

The D-1001 is a complete, stand-alone system when running the user program from PROM. However, during system startup, the program loader is attached by a cable and the dual processors share memories and control information. The controller uses a MOS Technology 6520 Peripheral Interface Adaptor [36] as a bi-directional communications port between the controller and the loader. The 6520 generates interrupts to the controller's microprocessor in response to the loader's requests for service. See Chapter 6 for a full description of its use.

4.2 Software

The operating system (OS) has the responsibility of managing system startup, power-fail restart, I/O snapshotting, interpretive execution of the user program, error detection, timer/counter arithmetic, interrupts from the timer clock and the program loader, and bi-directional communication with the program loader when programming or monitoring. The operating system is designed as a series of modules, each implementing

a separate and specific task. The programming is highly structured, highly modular, and well commented in an effort to localize responsibility for the various OS tasks. As a result, alteration of a particular module, whether it be due to a change in its operational definition or just error correction, is accomplished easily, quickly, with good reliability, and with a minimum of side effects on other related software modules. This structure is absolutely essential if we are to gain the full power of a microprocessor-based system; by so doing we gain maximum system flexibility by redefining system tasks as our needs become clearer and implementing those changes in software, long after the hardware decisions have been made and the unit has gone into production. The advantage of being able to make operational changes to the system by changing software instead of hardware has proven to be of such significant value that this fact alone justifies a microprocessor-based, software-driven system.

4.2.1 System Startup/Power Fail Restart

While initial system startup and a restart after a power failure could be (and usually are) handled by separate software modules, it was determined that in this particular environment they were so similar that they could be implemented using common code. This module's basic responsibility is to account for the fact that the most recent program solution (if there has been one) may no longer accurately reflect the status of the controlled machine (see Section 3.1.1).

The module is invoked by interrupt when the hardware detects a high-to-low transition on the RESET line. The $\overline{\text{RESET}}$ signal causes the CPU to stop execution at the end of the current instruction cycle, pick up a 16-bit address from dedicated locations $FFFE and $FFFF, and transfer that value to the processor's program counter, thereby effecting an unconditional jump to the address specified in the two-byte RESET vector. The RESET vector contains the starting address of the startup/reset module. The module immediately disables the recognition and servicing of any further interrupts occurring on the interrupt request line (IRQ), puts the processor in binary arithmetic mode, resets the stack pointer to the bottom of the system stack area, resets the one-second clock (software generated from the 0.1 second timer interrupts), and clears the system interrupt flags which indicate the occurrence of a timer interrupt or a request for service from the loader during a memory scan. At this point a decision must be made for each output and control variable as to whether or not it is retentive; if so, the variable should not be changed, otherwise it should be cleared. The module extracts a 127-bit vector from the compiled program (64 outputs and 63 control relays) and, depending on the value of each bit, either ignores or clears the corresponding output or control variable in the system I/O snapshot area.

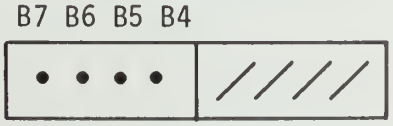
To avoid the condition of having timer and counter coils come up uninitialized, the initial state of timers and counters is clearly defined after startup/restart. Timer START coils are retentive, timer HOLD coils are cleared, counter RESET, count UP, and count DOWN coils are cleared, and both timer and counter OUTPUT contacts are retentive. The FIRST SCAN

coil is set to indicate that the user program will be in its first iteration. All such initialization requires approximately 4 ms. Finally, interrupts are enabled and control is transferred to the I/O snapshot module.

4.2.2 Input/Output Snapshot

The I/O snapshot module reads inputs from the real world, writes outputs to the real world, and updates the two copies of output and control variables which are necessary to guarantee "position independence." The snapshot records (1) the solution computed by the last program iteration (to be used as input to the forthcoming iteration) and (2) initializes the data area for the forthcoming iteration to be equal to the previous solution set. Thus, if an output is either set or reset by the previous iteration but its defining RLD page is skipped in the current iteration (perhaps a JUMP coil is active), its value will not change.

The I/O snapshot area occupies 127 4-bit bytes (retentive) in the system data area on page zero of the microprocessor's address space. The last scan and current scan values are stored in the most significant and the second most significant bits of the byte.



B7 - last scan value
B6 - current scan value
B5 - unused
B4 - unused

All software modules which use the status of inputs, outputs, and control variables interrogate only bit 7; the module which solves the RLD and generates new output solutions stores only to bit 6. This separation guarantees position independence while the program is in execution.

During the I/O snapshot, inputs which are on cause a '10' to be written to bits 7 and 6 of their respective data area; inputs which are off are treated as outputs. The status of each output is written to the real world based on the current value of bit 7, then bit 7 is set equal to bit 6. The tactic of having an active input report a '10' (currently on, soon to turn off) allows the snapshot module to correctly handle all 64 external inputs and outputs without ever forcing the user to provide a specific programming declaration of whether an individual terminal is an input or an output; rather, such definition is determined by each element's use. The D-1001 provides maximum flexibility for the user by allowing him to arbitrarily partition his 64 variable I/O space as best suits his problem. Thus, the user is not constrained to a maximum number of inputs or outputs, but only to a total of 64 I/O. The scheme is perfectly general and is easily expandable to any I/O capacity. The choice of 63 control relays was dictated by the limited amount of page zero retentive memory available and would be expandable given additional memory and a slightly longer execution time.

4.2.3 Counters

The counter software examines each of eight system timer/counter modules once per program iteration. The counters are unique

in that they are software edge-triggered (i.e., their last scan and current scan value must be '01' to be active). The advantage of edge-triggered operation is discussed in Section 3.1.1. By simulating the edge-triggered feature in software, four important benefits accrue: (1) we need not invest additional controller hardware (one-shots and flip-flops) for each of eight (possibly unused) counters; (2) the user avoids having to distinguish between timers and counters by any means other than his choice of keypresses; (3) the operating system can share the data area of counters (which are retentive) with the timers (which are not), and (4) the eight timer/counter modules may be allocated by the user as all timers, all counters, or any other combination as best fits the particular application.

If a timer/counter module was programmed as a counter, its RESET, count UP, and count DOWN coils and its OUTPUT contact are processed according to the flowchart in Figure 15.

Since the counter operations are primarily arithmetic, the counter module puts the processor in BCD arithmetic mode and proceeds to examine coils, increment and decrement counts, compare counts and setpoints, and determine outputs appropriately. The four BCD digits of the current count and setpoint are stored one digit per byte in the retentive CMOS memory, using the four most significant bits of each byte.

4.2.4 Timers

The timer module checks the timer interrupt flag to determine whether a timer clock interrupt occurred anytime during the last memory

For each counter:

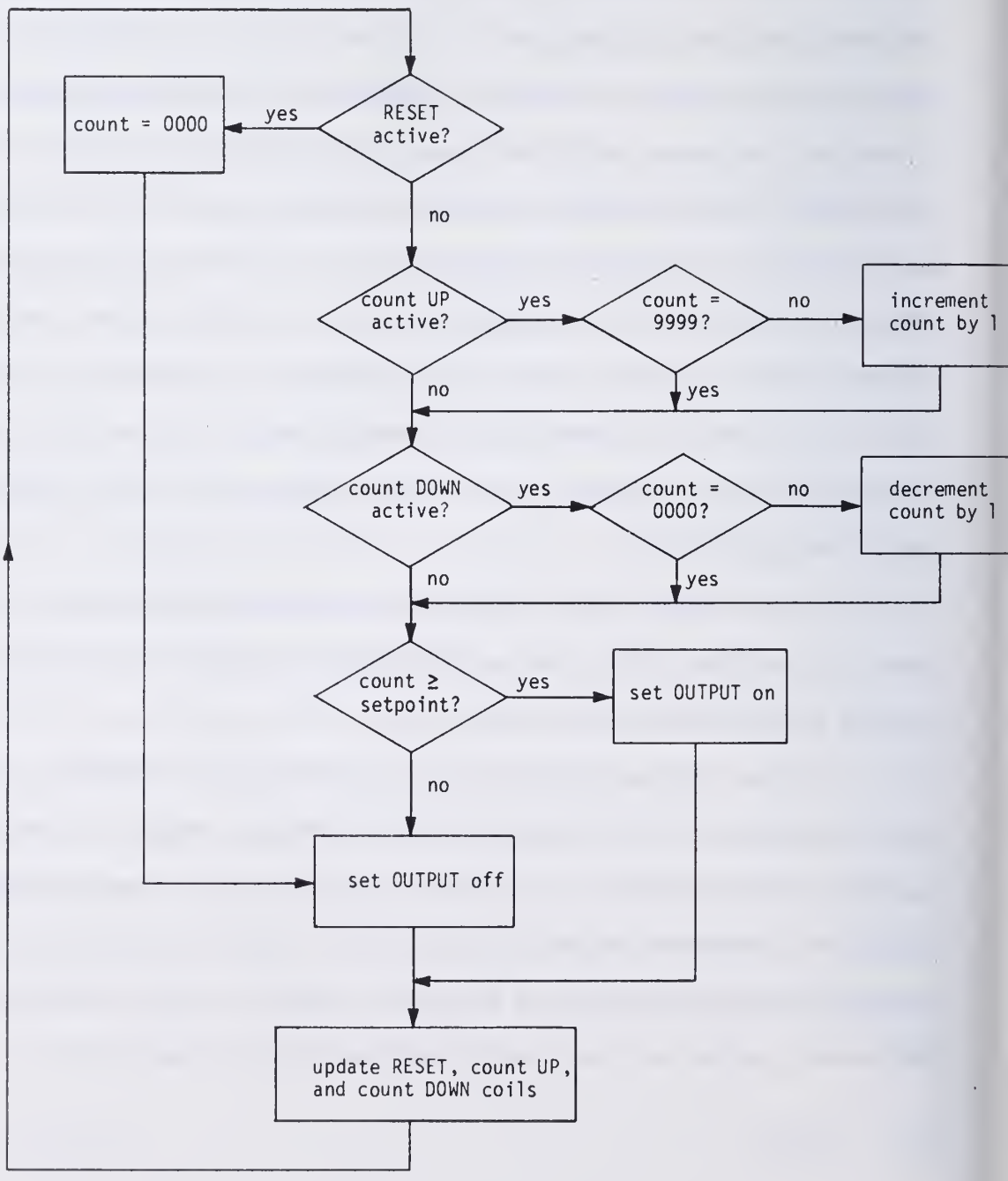


Figure 15. Flowchart of Counter Operation

scan. If not, an immediate exit is made and a new program iteration begun. If an interrupt did occur, this software module examines each timer/counter module, and, if it was programmed as a timer, processes its START and HOLD coils and its OUTPUT contact in accordance with the flowchart in Figure 16. Like counters, the timer module switches to BCD arithmetic mode for examining, incrementing, and comparing the 4-digit BCD current counts and presets. Unlike counters, none of the variables are edge-triggered so that a timer, once turned on, will continue to time until either it has timed out (count equals zero) or it is reset.

4.2.5 Column Evaluator

The compiled code for a single column of relay symbols begins with a one-byte code which selects one of sixteen submodules to process the remaining column code. Since the majority of the system's time (approximately 75 percent) is spent evaluating relay columns, the code for such evaluation is highly optimized for speed, even at the cost of space for redundant code. The selection of the proper processing unit is made through a jump table which selects the proper submodule or entry point to be used for the evaluation (for instance, the code for processing column pattern #7 with relays on lines 2, 3, and 4 is an entry point in the module which processes column pattern #15).

Regardless of the subunit doing the processing, the algorithm is the same. For each relay in the chosen pattern, in order from bottom to top, an I/O address is read from the macrocode and the status of the element at that address fetched from the snapshot area, resulting in a

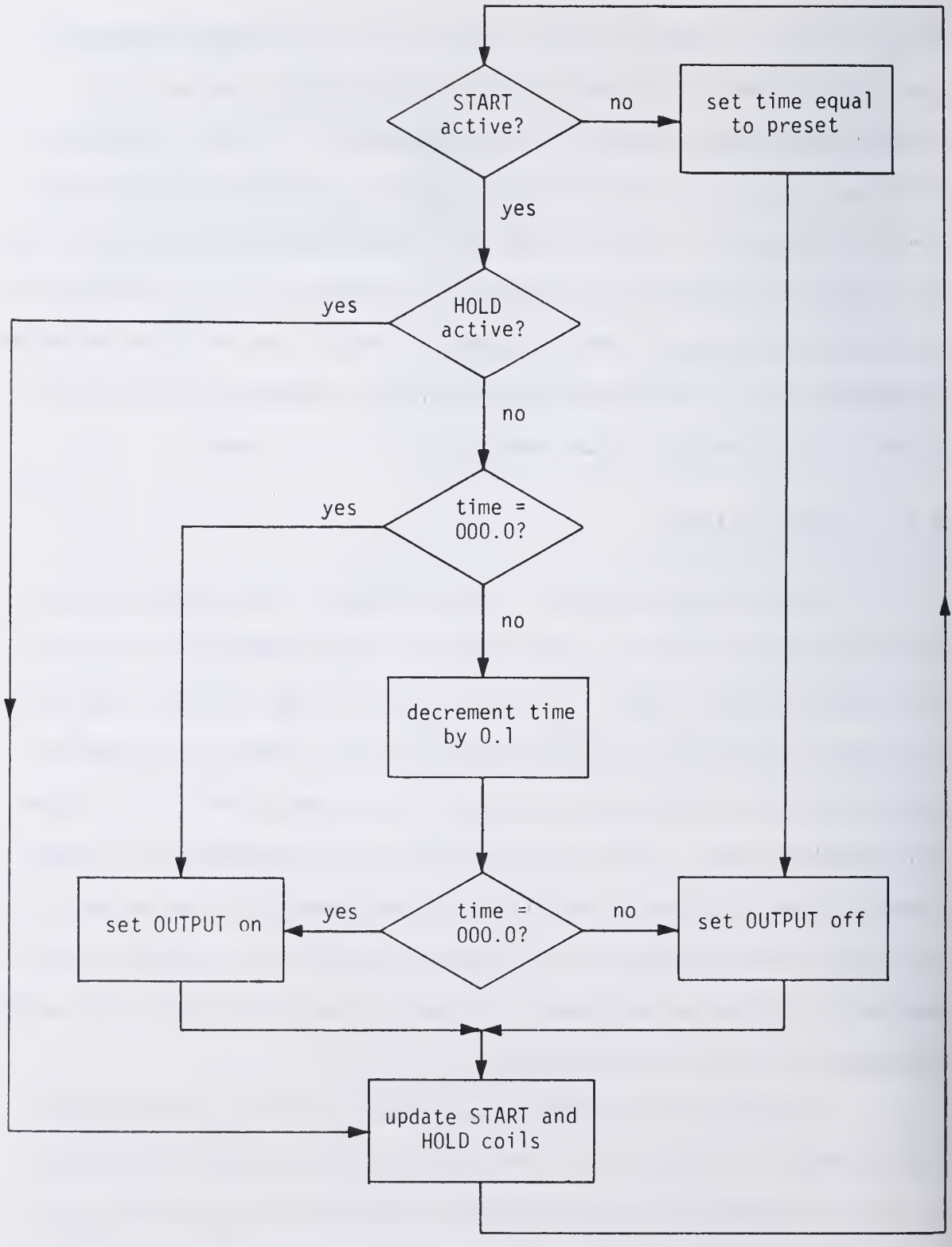


Figure 16. Flowchart of Timer Operation

4-bit vector representing the status of the elements on the four horizontal lines (nonexistent elements produce a '0'). The logical AND of this vector with another containing the conductive status of the column's left-hand nodes yields a temporary variable which would describe conduction through the column if all relays were of the normally open type. The last byte of code for any relay column specifies which, if any, of the horizontal lines contain complemented elements and which verticals are present. Exclusive-ORing the temporary variable with this "complement/vertical" byte yields a compact, 7-bit code which identifies this specific instance of column conduction (see diagram in Section 3.5) as a function of fifteen variables: the column's left-hand nodes N_1, N_2, N_3, N_4; the conductive status of the column elements R_1, R_2, R_3, R_4; the element complements C_1, C_2, C_3, C_4; and the verticals V_1, V_2, V_3.

Having built the bit vectors $\overline{N}$, $\overline{R}$, $\overline{C}$, and $\overline{V}$ as described, the solution to the column conduction problem, which yields $\overline{N}' = f(\overline{N},\overline{R},\overline{C},\overline{V})$, is accomplished by using the 7-bit code as an index into a 128-byte lookup table; the lookup requires a mere 23 μs.

Having the 4-bit $\overline{N}'$ column conduction vector for any column makes checking for total nonconduction simple. If $\overline{N}' = 0000$, the user program counter which originally pointed to the optimization byte (the "output column pointer" in Section 3.5) is added to the content of the optimization byte and stored back into the user program counter, thereby effecting a jump across any and all intervening relay columns and resuming execution at the page's output column with $\overline{N} = 0000$. Output column processing (including possible complementation) then proceeds normally. This checking for a "dead" page is done automatically and efficiently at the end of

each column conduction computation. The overhead is extremely low (simply a test for $\overline{N}' = 0000$) and the expected gain in execution speed is statistically quite high (see Section 3.3).

The module which evaluates output columns is similar in function, but is optimized for code space rather than time since the number of relay columns is expected to outnumber the output columns by a factor of five, and also because the number of combinations of output symbol types and positions is much larger than for relay columns. As before, the first byte of the output column code selects the system module which processes output columns; the second byte ("complement/pattern") identifies which, if any, outputs are complemented (i.e., normally closed relays) and from which lines they were stored. Each '1' bit in the pattern code causes an output address to be read from the macrocode and the corresponding conduction value stored.

The special JUMP and RESET coils are defined as special columns and are treated similarly to output columns. If the JUMP coil is active, the user program counter is repeatedly advanced to the next page mark until the required number of pages have been skipped. The RESET coil accomplishes the same type of skipping, but also resets every output address found on the pages it passes over.

4.2.6 Error Conditions

If, due to an error in the compiled code or random electrical noise, the operating system's internal program counter or other critical system data were to be in error, the interpreter cannot recover gracefully.

Two provisions are made for this eventuality. First, each page of the RLD code is preceded by a special one-byte "page mark" which identifies the beginning of a new page and the establishment of the left-hand power line. Should the interpreter not find a page mark at the end of each output column's code, it will stop execution, record the error, and restart at the beginning of the user's program. If such an error occurs twice during the same memory scan, an error module is invoked which turns off all real world outputs, lights an LED error signal on the front panel, and shuts down the controller to await a manual restart.

Second, it is possible that a random disturbance could alter system data and result in an infinite loop in the operating system itself. To detect this eventuality, the operating system generates a 5 μs pulse once per scan (depending on program length, every 5 to 20 ms) on a dedicated output port. Should the pulse fail to appear for 0.1 seconds, outboard hardware signals an error and initiates an automatic restart.

4.2.7 Interrupt Handler

If interrupts are enabled, the appearance of an interrupt signal on the interrupt request line causes the microprocessor to stack its current program counter and processor status word and branch to the address contained in the IRQ interrupt vector at locations $FFFC and $FFFD. The interrupt handler checks control register B of its PIA to determine if the interrupt was originated by an attached program loader. If so, the request for service is noted and some additional information saved; if not,

the interrupt must have been generated by the timer clock and that event is noted in the timer interrupt flag. No other processing occurs during the interrupt to insure that no significant data changes state during the course of a single program evaluation. All interrupt-requested services are performed between program iterations, as discussed in Chapter 6.

4.2.8 Communications

An attached program loader can force the controller into a number of different operating modes, either singly or in combination. The loader can monitor any one address in the controller, monitor all of the system data area, insert new timer/counter presets, examine all timer/counter current counts and presets, override selected inputs by forcing them on or off, or substitute a user program in RAM for one in PROM. The controller services all such requests for service between memory scans as explained in Chapter 6.

5. THE DIRECTOR 1001 PROGRAM LOADER

5.1 Hardware

5.1.1 The Microprocessor

For essentially the same reasons that the MOS Technology 6502 microprocessor was chosen as the basic intelligence of the controller, the same CPU was chosen as the control unit of the program loader. Although the responsibilities of the loader are quite different from those of the controller, the need for moderate speed and a powerful instruction set still predominate. While the controller runs continuously at full speed, solving and resolving the RLD as fast as possible, the loader operates more nearly in "burst mode" in response to keypresses. While programming, the processor experiences a 90 percent idle time as it collects and processes keypresses and updates the CRT display; when monitoring a running controller, the CPU must interface interactively with the controller's processor and idle time drops to about 10 percent. Thus, in this case the loader needs the same speed characteristics as the controller. Likewise, the loader's operating system benefits tremendously from the host of addressing modes provided by the 6502, as it performs a wide variety of system tasks.

Using the same microprocessor in both units is also advantageous in terms of reducing the number of different components which must be bought, stocked, and serviced. Much of the power supply and interface circuitry are the same in both loader and controller.

5.1.2 The Memories

Like the controller, the loader must use several different types of memory to perform its various functions. The memories required include a ROM for the operating system, a RAM for the storage of system variables and the encoded graphic program, a RAM for the user program used during system startup and debugging, and yet another RAM used for communicating status information to and from the controller.

5.1.2.1 Operating System ROM

The operating system is relatively complex and requires approximately 8K to implement the various OS tasks, including keyboard decoding, CRT display management, graphic program preparation and editing, translation of the RLD into internal code and vice-versa, monitoring of a running controller, forcing selected inputs on and off, data transfers among PROM, RAM, and an external RS-232C port, and PROM/RAM program equivalence verification. While the operating system currently occupies eight 1K x 8 Intel 2708 PROMs, it will ultimately be committed to ROM.

5.1.2.2 Program RAM

Although the controller will normally be interpreting the compiled user program from PROM, a 1K program RAM (expandable to 2K) is available in the loader for use during system startup and debugging. The primary advantage is that simple program changes can be made wholly within the loader environment, without the necessity for programming a new PROM

chip (a 3 minute wait) or physically transferring the PROM between machines.

5.1.2.3 Graphic Storage RAM

A 4K RAM provides storage for all the operating system's internal variables and for an encoded (but not compiled) form of the graphic RLD. Each RLD page is reduced to a 96-byte master template which completely describes every matrix element and I/O address used on the page. Editing of any RLD page involves extracting its associated master template and decoding it into the original relay symbols and I/O addresses. Since the template is fixed-length, page editing, replacement, insertion, and deletion result in data movement within the graphic storage RAM, but require no complicated dynamic memory allocation or garbage collection.

5.1.3 PIA Communications

The loader uses two PIAs to exert control over its environment. One provides PROM/RAM chip enable signals, software memory protection signals, control information for the CRT, and actuates an audible error signal (beeper). A second PIA is used for asynchronous communications with the controller and provides memory switching signals for the communications RAM and user program RAM, generates interrupts to the controller, receives interrupt acknowledge signals from the controller, and transmits status information through its B data port. The PIA's bi-directional data ports and interrupt generation facilities are wholly under software control.

5.1.4 Front Panel

The front panel is shown in Figures 17a and b. The layout of the panel has been the object of considerable human engineering in an effort to make the "keypress programming" as simple, logical, and unambiguous as possible. Figure 17a shows the primary programming keyboard which is mounted almost horizontally on the front panel (much like a typewriter keyboard); Figure 17b shows the CRT display screen and mode selection control buttons. The latter panel is mounted above the programming keyboard at a 60 degree angle from the horizontal to afford good visibility to the user.

5.1.4.1 Keyboard

The keyboard consists of 58 individual keys, physically subdivided into ten separate groups: a ten-digit keypad for inputting I/O addresses and timer/counter presets, six relay element symbols, five coil symbols, five cursor control keys, two RLD page selection keys (scroll forward and backward), seven function keys for EDIT mode, five function keys for MONITOR mode, four keys which direct memory transfers, six keys for primary mode selection, and a three-position memory protection switch.

Each of the individual keys are normally-open, momentarily-closed pushbutton switches. The keys are logically laid out as an 8 x 8 matrix, as shown in Figure 18, and keypress decoding is accomplished by software interrogation of the two 8-bit latches. Each keypress generates an interrupt and makes a momentary connection between one row and one column. By writing all ones to latch A and reading latch B, the software

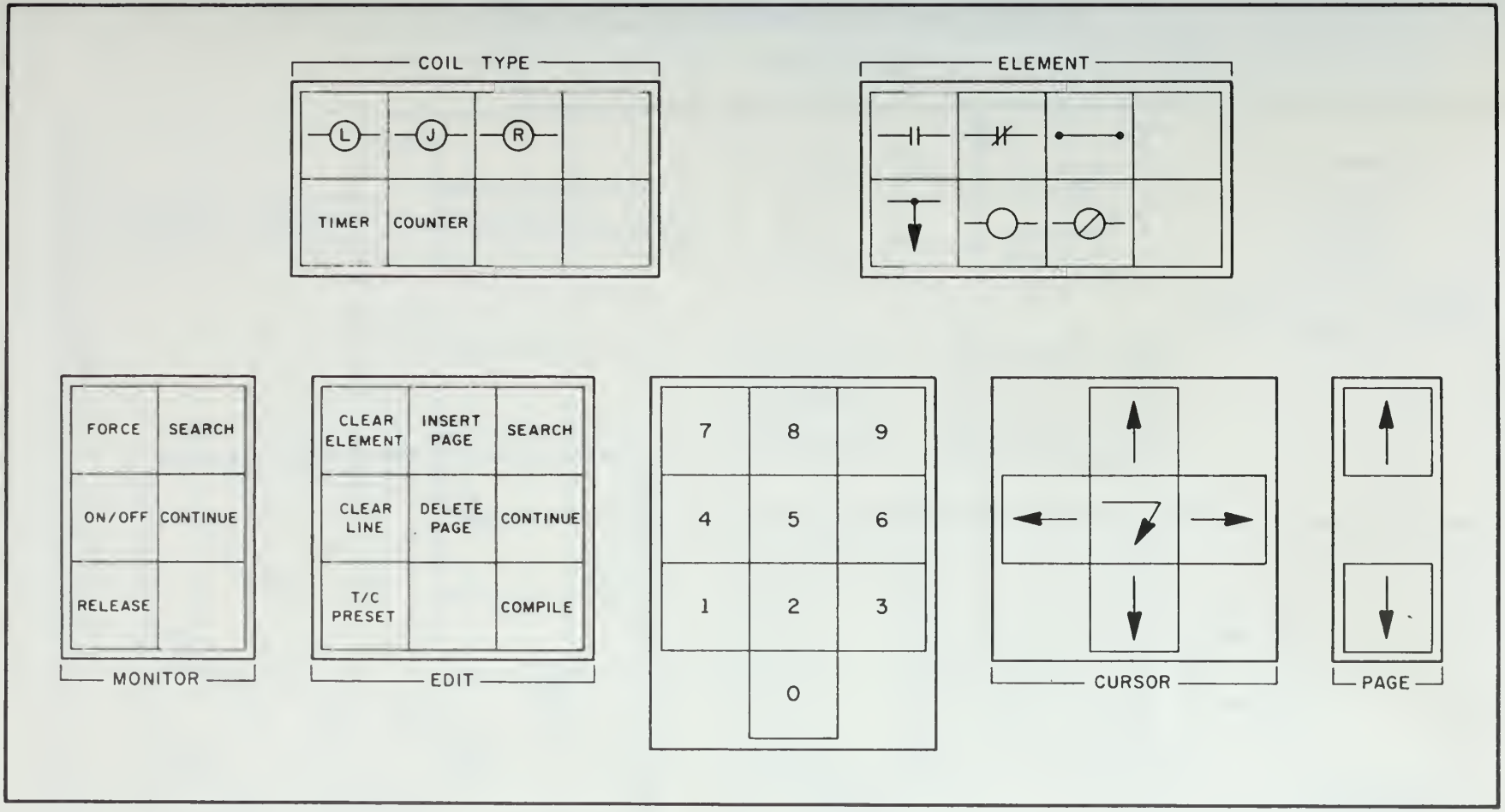


Figure 17a. Programming Keyboard

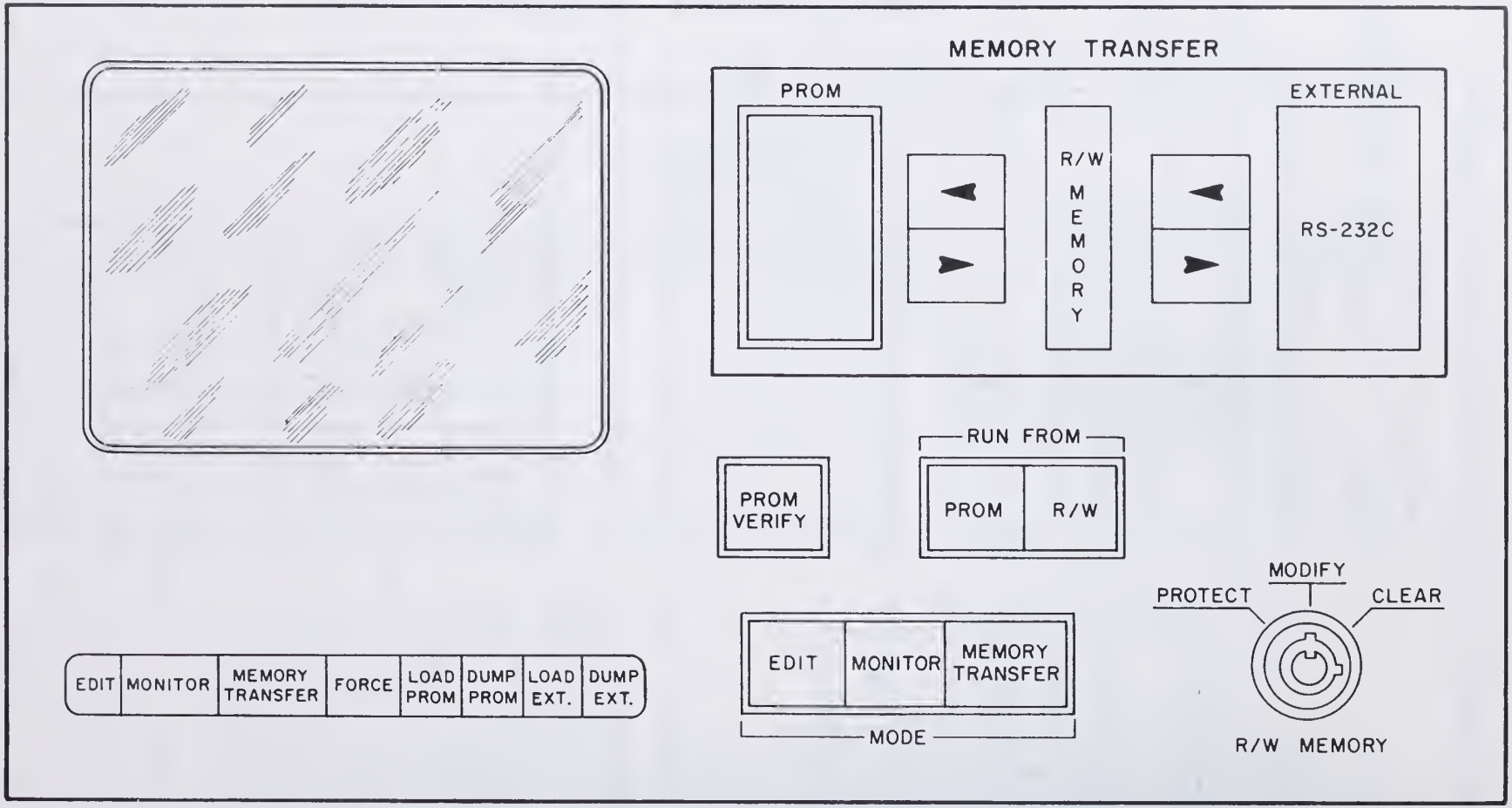


Figure 17b. CRT and Mode Selection Keyboard

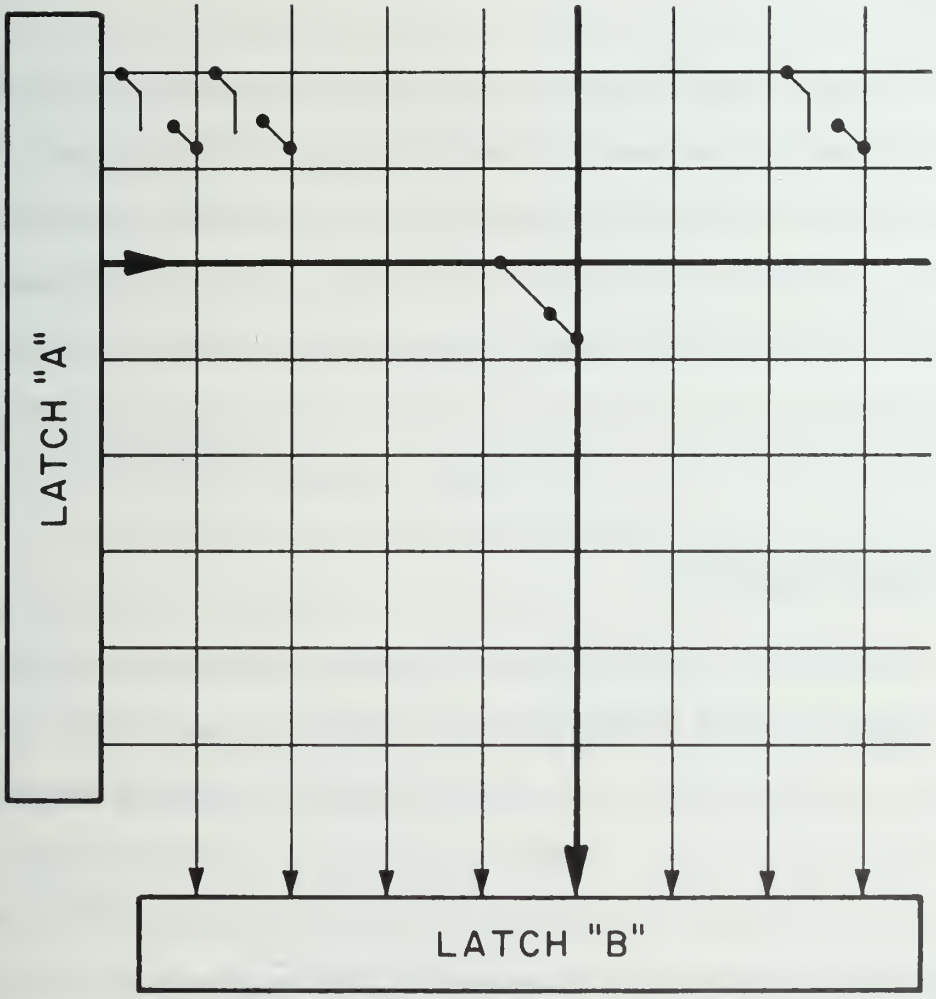


Figure 18. Keyboard Encoding Matrix

can determine in which logical column the keypress occurred; similarly, writing ones to latch B and reading latch A identifies the row. From the row and column information the individual key is uniquely determined.

5.1.4.2 CRT

The relay ladder diagram is graphically displayed on a 6-inch CRT screen located in the upper left-hand corner of the front panel. The alphanumeric CRT can display any standard ASCII character in any position of its 32 character per line, 16 line screen. During programming and editing, the current page number is continuously displayed in the upper right hand corner of the screen and a blinking cursor identifies which of the 32 matrix positions is currently being programmed/edited.

5.1.4.3 External Connector

The user will ultimately need hard-copy documentation of his RLD. The memory transfer pushbuttons and external connector allow the user to attach any peripheral device obeying RS-232C protocol (teletype, cassette tape, computer, etc.). Depending upon the operational mode, the loader can output either the ASCII character representation of the RLD program (as seen on the CRT) or the machine code image of the compiled program.

In addition to outputting documentation, the loader will also accept input from the outside world, thereby enabling the loader to be programmed as an external device. Although not initially implemented, the

design of the external port is sufficiently general to permit limited data acquisition from, say, a central computer facility.

5.1.4.4 Memory Protection

A three-position read/write memory protect switch prevents accidental erasure of an entire RLD program. In its "protect" position, the switch disables the WRITE line to the graphic storage RAM, making it a read-only memory; in "modify" position the switch enables writing to the memory for programming and editing; in "clear" position (momentary) the hardware clears the whole memory.

5.1.4.5 Mode Lights

Eight LEDs are located under the CRT and, via software control, are illuminated underneath a silkscreen overlay to show in what mode(s) the loader is currently operating. The individual modes of operation include: EDIT (same as PROGRAM), MONITOR, FORCE, MEMORY TRANSFER, LOAD PROM, LOAD EXTERNAL, and DUMP EXTERNAL.

5.2 Software

The operating system is responsible for the coordination of the entire program loader system and its interaction with the controller. As we found with the controller's software, the ability to configure the hardware and commit it to production, and to later change the entire character of the programming system by merely changing the system software, has proved to be such a significant advantage to the development of the

total system that it justifies all the effort and complexity of a software-driven system.

The loader's operating system, like that of the controller, is a highly structured, highly modular program. Because the majority of OS tasks face no serious execution time requirements, its coding tends to be space-optimal rather than time-optimal, resulting in a multiplicity of general purpose, parameterized subroutines.

The operating system supports three distinct modes of operation (program/edit, monitor, and force) and a number of utility functions (e.g., memory transfer, PROM/RAM verify, external I/O) as discussed below.

5.2.1 PROGRAM/EDIT Mode

The basic function of the program loader is to provide the facility for graphic programming. When the user wishes to generate a program, he need not be near the controller, because the loader is a stand-alone unit. The user may program it anywhere. In fact, it is never necessary for the controller and loader to be connected, only useful. The user can conduct all his programming activities in his office and carry only the resulting PROM chip out to the controller.

PROGRAM and EDIT mode are essentially the same; editing only implies the previous existence of a graphic program. Cycling the main power switch or moving the memory protection switch into its CLEAR position will erase any program in the RAM. Programming amounts to editing a blank program.

When the EDIT mode is selected, the CRT displays the first page of an RLD program, or a blank page if the memory has been cleared. A

blinking cursor is positioned at the upper left corner of the screen, indicating that the RLD matrix position just beneath it may now be programmed. Depressing any of the relay symbol keys will cause its associated picture to be displayed. Any number of relay keys may be depressed sequentially; the display changes with each keypress so that only the last keypress in any one matrix position has any effect. This feature makes error correction of a single element trivially easy--if the wrong symbol is entered, entering the correct one automatically replaces the erroneous one. Depressing the digits of the keypad alter the displayed symbol's I/O address by shifting its currently displayed address one digit to the left and inserting the new digit in the least significant digit position. When any one matrix position has been programmed, the five cursor control keys allow the user to move the screen cursor one element to the left or right, forward or backward one line, or forward to column one of the next line.

Since output coils are restricted to column eight, depressing any of the coil symbols automatically advances the cursor to the end of the current line and inserts the coil symbol in column eight. Any blank elements passed over on the line are changed into horizontal connections. In the normal case programming logic tends to cluster in the upper left corner, so this feature eliminates the need for multiple keypresses to advance the cursor to the output column. Thus, the relay symbols, coil symbols, I/O address keypad, and cursor control keys provide full programming/editing control over a single page.

Two additional keys located to the left of the cursor controls permit scrolling forward or backward through an existing program. Each down-arrow keypress advances the display to the next RLD page; each up-arrow keypress moves the display back one page. The response time for page scrolling is immediate, so the user can browse through a 30-page program in less than a minute.

The TIMER/COUNTER PRESET pushbutton is used for specifying presets and setpoints when they are known in advance. Depressing this key replaces the current RLD page display with a menu of all timer/counter I/O addresses and their current presets/setpoints (initially zero). The cursor control keys may then be used to position the screen cursor at the desired line (timer/counter numbers 1 - 8); the keypad is used to insert or change any preset/setpoint.

In addition to page-by-page scrolling, the EDIT mode supports a search function. Pressing the SEARCH key allows the user to specify a particular symbol and I/O address; pressing CONTINUE causes the system to search the entire graphic program, beginning at page one, and display the first page on which the specified combination of symbol and address is found. Pressing CONTINUE again repeats the search, beginning on the next relay page. By using SEARCH and CONTINUE a user can very quickly locate all instances of a relay symbol and address in a program. The search mode is terminated when either the specified element is not found (a message to that effect is displayed on the screen) or when the EDIT pushbutton is depressed.

Because no translation of the graphic program occurs until the COMPILE pushbutton is depressed, it is possible that a very long RLD would generate more than 1K of code (the PROM chip capacity). Should this happen, an audible error signal is emitted and an error message displayed. Errors in the programming syntax (such as attempting to put a relay symbol in the output column, inserting a vertical connection below the bottom line, or using an invalid I/O address) generate three responses: (1) the error beep is emitted; (2) the invalid keypress is ignored; and (3) the cursor controls become inoperative until the error is corrected. Thus, it is impossible to generate a syntactically incorrect program.

5.2.2 MONITOR Mode

Having generated a syntax-error-free program, it may now be tested by a running controller. The monitor mode permits the user to scroll through a graphic program, select any page, and watch the relay contacts open and close in real time. By interacting with a cable-connected controller, the loader generates a request for a copy of the most recent I/O snapshot. Between program solutions the controller stores a copy of this I/O snapshot in the communications RAM, which, while physically located in the loader, can be logically switched into and out of the controller's address space. Having obtained the current status of all inputs, outputs, control variables, etc., the loader alters the display to show which elements are conducting. A normally-open relay element which is currently energized, or a normally-closed element which is currently deenergized, is shown to be conducting by the placement of a

three-character-wide solid bar above its relay symbol in the display. Likewise, outputs which are energized by the currently displayed page of relay logic are shown to be active by the solid bar above their coil symbol (the bar is shown for normal outputs which are energized and for complemented outputs which are deenergized).

In a normal program, any I/O address may be used many times on many different pages of the program. The search function described in the previous section is useful for monitoring the influence of a particular element on many relay pages.

5.2.3 FORCE Mode

One typical problem which occurs during the installation of a new control system is that some device on the factory floor fails to respond as required. The immediate question is: is this a hardware failure of the controlled device itself, or is this due to a programming error? The traditional approach has been to isolate the elements which participate in the control of the device (limit switches, pushbuttons, other outputs, etc.) and manually turn them on or off (as required by the programming logic) to guarantee a conducting path through the matrix activating the device in question. If the device fails to respond, the trouble is likely to be in the device itself (stuck relay, burned-out light, etc.); otherwise the trouble could be a programming error. This determination requires a significant investment in both time and troubleshooting talent.

The loader deals with this situation by offering a MONITOR mode in which the dynamic status of all control elements can be examined, and a

FORCE mode in which the status of all selected inputs can be overridden from the loader's front panel. As with EDIT mode, the user may scroll through his program, or SEARCH, until he has located the page of relay logic which controls the device in question. He enters FORCE mode by pressing the FORCE pushbutton. By positioning the screen cursor over any element, the user may now override any input's real world value by pressing the ON/OFF pushbutton. Each time ON/OFF is depressed, the loader commands the controller to substitute, via software, a forced value for the specified input's real world status. Each ON/OFF keypress toggles the value to be forced from on to off or off to on. Any or all inputs may be forced. The forced condition is removed from any element when the cursor is positioned over that element and the RELEASE button is depressed. FORCE mode is canceled entirely when any other mode is selected.

5.2.4 VERIFY Mode

The user may verify that a compiled program in PROM is identical to the one in RAM by pressing the VERIFY key. The software simply compares the two 1K programs and determines whether or not they are the same, announcing the result on the CRT. This function is especially useful if memory transfers are being done on the factory floor where random electrical noise might cause otherwise undetected I/O errors.

5.2.5 MEMORY SELECT

The controller can run the user's program from either its own on-board PROM or from the loader's 1K program RAM. The "RUN FROM RAM"

switch causes the loader to switch its program RAM into the controller's address space and to instruct the controller to "run from RAM." This feature is especially useful during system startup when program changes are frequent; it avoids the time delay of producing a new PROM after every compilation and transferring the PROM between machines.

5.2.6 Memory Transfer

The operating system provides four-way memory transfer among the PROM, RAM, and external I/O port. The labeled arrows on the "memory transfer" pushbuttons indicate in which direction the transfer is to occur. Transfer of the 1K compiled program from PROM to RAM is immediate; transfer from RAM to PROM requires three minutes (the nature of the PROM and its programming hardware is such that all 1K locations must be individually written 100 times). For purposes of documentation, either the compiled program or the ASCII character RLD pages may be printed by attaching an appropriate RS-232C compatible device (e.g., teletype, computer) to the external I/O port. In EDIT mode the RAM-to-external button causes the RLD picture to be transmitted; otherwise, the compiled code is output.

A useful, but as yet unimplemented, feature is the external-to-RAM transfer which will allow loading of a program from an external device such as a cassette tape or, ultimately, another program loader or a central computer.

6. CONTROLLER-PROGRAM LOADER COMMUNCATIONS PROTOCOL

6.1 Overview

The controller and program loader communicate by:

1. Passing information through a switchable 4K address space (containing the RAM version of the user program);
2. Passing information through a switchable 512-byte address space (including monitor results, force mode instructions, timer/counter presets, etc.); and
3. Using two PIAs (MOS Technology 6502s), one each in the controller and program loader, to signal requests for service and to switch and synchronize the memories.

The master layout of the communications hardware is shown in Figure 19. The two PIAs are directly addressable only in their respective address spaces; however, their 8-bit B data registers are bi-directional and hard-wired together so that 8 bits of control information can be passed in either direction. Additionally, the CB2 line of each PIA can, under software control, generate interrupts depending upon the status of the PIA's control registers. As detailed later, a common operation will be for the controller to set its PIA for input, the loader will set its PIA for output, the loader will set a code in its B data register and interrupt the controller, and the controller will respond with the requested service.

Figure 19 also shows the layout for the 512-byte memory; the layout for the 4K address space is almost identical (the switching hardware

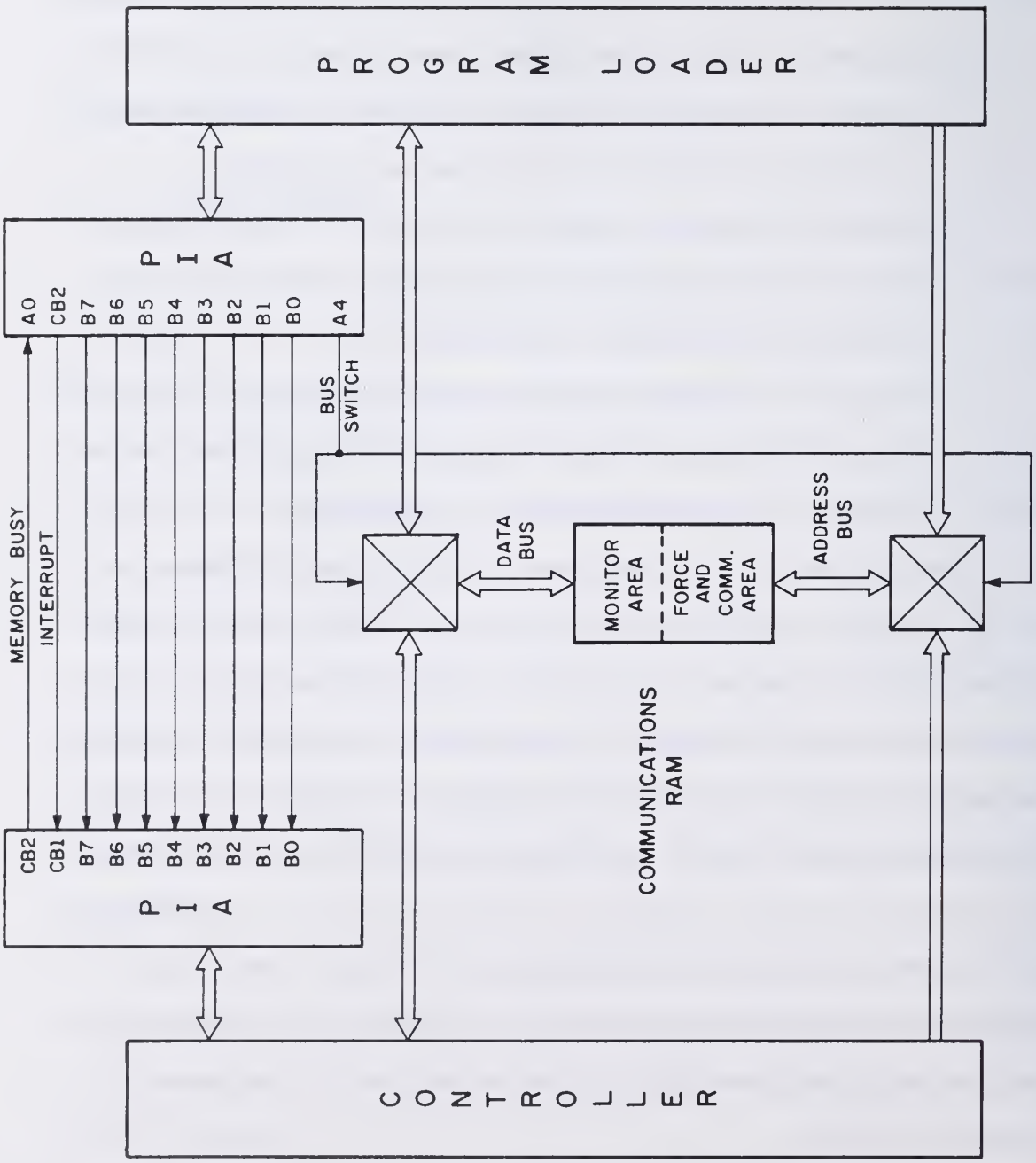


Figure 19. PIA Interconnections

is controlled by a different bit in the loader PIA's control register A).

To avoid confusion, the 4K address space will hereafter be called the "program RAM" (as distinguished from the "program PROM" in the controller); the 512-byte memory will be called the "communications RAM."

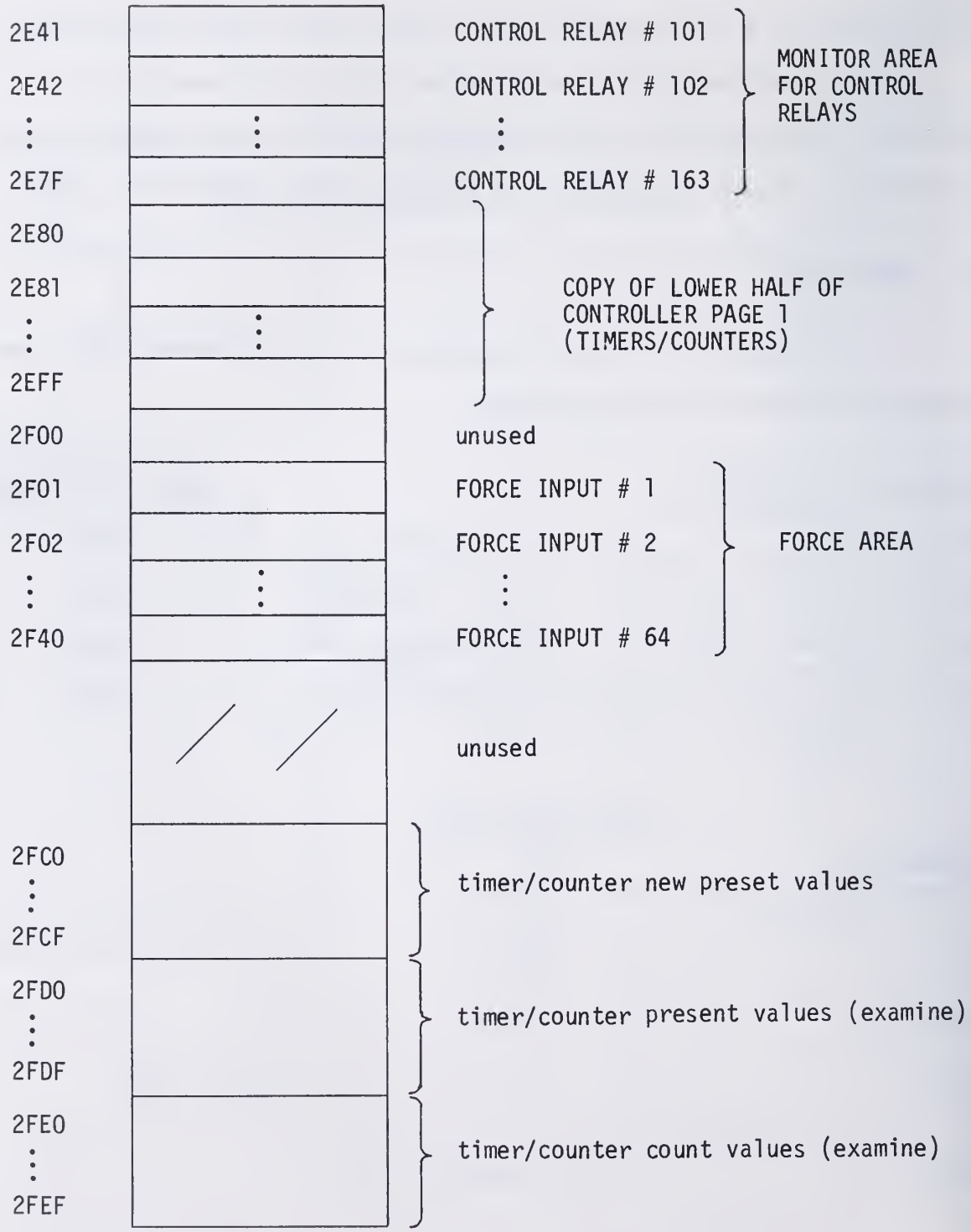
6.2 Address Map

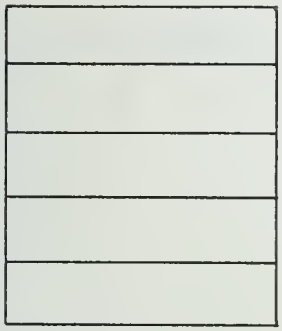
Shown below is a map of the address space used by the PIAs, the communications RAM, and the program RAM.

Controller PIA Address		Loader PIA Address
4014	data register A	41FC
4015	control register A	41FD
4016	data register B	41FE
4017	control register B	41FF

Communications RAM

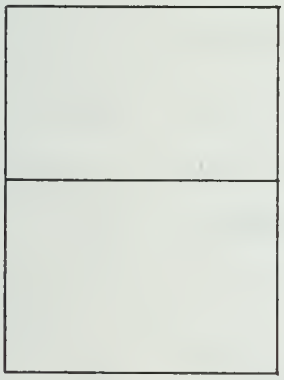
Hexadecimal Address (common to controller and loader)	Content	
2E00	unused	
2E01	I/O #1	MONITOR AREA FOR I/O
2E02	I/O #2	
⋮	⋮	
2E40	I/O #64	


2E41
2E42
2E7F
2E80
2E81
2EFF
2F00
2F01
2F02
2F40
2FC0
2FCF
2FD0
2FDF
2FE0
2FEF
CONTROL RELAY # 101
CONTROL RELAY # 102
CONTROL RELAY # 163
MONITOR AREA FOR CONTROL RELAYS
COPY OF LOWER HALF OF CONTROLLER PAGE 1 (TIMERS/COUNTERS)
unused
FORCE INPUT # 1
FORCE INPUT # 2
FORCE INPUT # 64
FORCE AREA
unused
timer/counter new preset values
timer/counter present values (examine)
timer/counter count values (examine)



2FF0	timer/counter preset flag bits
. . .	unused
2FFD	monitor address (low byte)
2FFE	monitor address (high byte)
2FFF	monitor address (8 bits)

Program RAM



3000 . . . 33FF	1K RAM for user program
3400 . . . 3FFF	reserved for future expansion

Communications RAM (2E00 - 2FFF)

Monitor Area (2E00 - 2FFF)

Hexadecimal Address	Content	Valid Bits
2E01	I/O # 1	B7
2E02	I/O # 2	B7
. . .	. . .	. . .
2E40	I/O # 64	B7

Address	Content	Valid Bits
2E41	CONTROL RELAY # 101	B7
2E42	CONTROL RELAY # 102	B7
.	.	.
.	.	.
.	.	.
2E7F	CONTROL RELAY # 163	B7
2E80	t/c #1 output	B7
.	.	.
.	.	.
.	.	.
2E87	t/c #8 output	B7
2E88	timer #1 start/counter # 1 reset	B7
.	.	.
.	.	.
.	.	.
2E8F	timer #8 start/counter #8 reset	B7
2E90	timer #1 hold/counter #1 up	B7
.	.	.
.	.	.
.	.	.
2E97	timer #8 hold/counter # 8 up	B7
2E98	counter #1 down	B7
.	.	.
.	.	.
.	.	.
2E9F	counter #8 down	B7

Address	Content	Valid Bits
2EA0	t/c #1 preset MSD	B7,B6,B5,B4
2EA1	t/c #1 preset 2nd MSD	B7,B6,B5,B4
2EA2	t/c #1 preset 3rd MSD	B7,B6,B5,B4
2EA3	t/c #1 preset LSD	B7,B6,B5,B4
.	.	.
.	.	.
.	.	.
2EBC	t/c #8 preset MSD	B7,B6,B5,B4
2EBD	t/c #8 preset 2nd MSD	B7,B6,B5,B4
2EBE	t/c #8 preset 3rd MSD	B7,B6,B5,B4
2EBF	t/c #8 preset LSD	B7,B6,B5,B4
2EC0	t/c #1 count MSD	B7,B6,B5,B4
2EC1	t/c #1 count 2nd MSD	B7,B6,B5,B4
2EC2	t/c #1 count 3rd MSD	B7,B6,B5,B4
2EC3	t/c #1 count LSD	B7,B6,B5,B4
.	.	.
.	.	.
.	.	.
2EDC	t/c #8 count MSD	B7,B6,B5,B4
2EDD	t/c #8 count 2nd MSD	B7,B6,B5,B4
2EDE	t/c #8 count 3rd MSD	B7,B6,B5,B4
2EDF	t/c #8 count LSD	B7,B6,B5,B4
2EF0	first scan	B7
2EF1	one second clock	B7

Force Area (2F01 - 2F40)

	Address	Content	Valid Bits
	2F01	force input #1	B7,B6
	2F02	force input #2	B7,B6
	.	.	.
	.	.	.
	.	.	.
	2F40	force input #64	B7,B6

Timer/Counter Area (2FC0 - 2FF0)

PRESET SET	2FC0	t/c #1 preset MSD/2nd MSD	all
	2FC1	t/c #1 preset 3rd MSD/LSD	all
	.	.	.
	.	.	.
	2FCE	t/c #8 preset MSD/2nd MSD	all
	2FCF	t/c #8 preset 3rd MSD/LSD	all
PRESET EXAMINE	2FD0	t/c #1 preset MSD/2nd MSD	all
	2FD1	t/c #1 preset 3rd MSD/LSD	all
	.	.	.
	.	.	.
	2FDE	t/c #8 preset MSD/2nd MSD	all
	2FDF	t/c #8 preset 3rd MSD/LSD	all
COUNT EXAMINE	2FE0	t/c #1 count MSD/2nd MSD	all
	2FE1	t/c #1 count 3rd MSD/LSD	all
	.	.	.
	.	.	.
	2FEE	t/c #8 count MSD/2nd MSD	all
	2FEF	t/c #8 count 3rd MSD/LSD	all
	2FF0	t/c preset flag bits	all

Monitor One Address Area (2FFD - 2FFF)

2FFD	monitor address (low byte)	all
2FFE	monitor address (high byte)	all
2FFF	monitor results (8 bits)	all

6.3 Interrupt Service

A request for service is always initiated by the program loader. It is the responsibility of the loader to have the switchable memories correctly switched and the mode code correctly set before generating an interrupt to the controller.

When interrupted, the controller will recognize the interrupt condition and determine whether the interrupt was caused by the 0.1 second clock or by the loader, via its PIA. This determination is accomplished by having the controller check the control register of its PIA to see if its interrupt bit is set.

If the interrupt was caused by the clock, system timers will be processed normally. If the interrupt was caused by the PIA, an 8-bit mode code will be read by the controller from PIA data register B, but no action will be taken by the controller until the current memory scan is complete. Thus, the controller will recognize interrupts as they occur, but will process them only between memory scans. Software provisions are made to handle the situation of having a timer clock interrupt occur while a PIA interrupt is being serviced.

The 8 bits of the mode code are assigned as follows:

PIA DATA REGISTER B

B7	B6	B5	B4	B3	B2	B1	B0

B7: monitor all

B6: monitor one address

B5: run from RAM

B4: force

B3: set timer/counter

B2: examine timer/counter

B1: copy program

B0: unused

6.4 Memory Switching

The program RAM and the communications RAM have separate switching hardware, each exclusively under control of the loader. The address space to which the program RAM is connected is controlled by loader PIA data register A, bit 5 (0 = loader, 1 = controller). Likewise, the address space to which the communications RAM is connected is controlled by loader PIA data register A, bit 4 (0 = loader, 1 = controller).

Synchronization of the program RAM is simple since it is read-only for the controller. Thus the loader need only insure that (1) the program RAM has been switched into the controller's address space before initiating "run from RAM" mode, and (2) that "run from RAM" mode has been terminated before the program RAM is switched back to the loader.

Sharing the communications RAM is a more difficult task to synchronize since both the controller and loader need it. If, for

instance, the loader wishes to initiate monitor mode, it must give the monitor area to the controller long enough to get a complete copy of the I/O snapshot, but then the loader must regain control of the monitor area so it can read the results. This synchronization is accomplished by having the loader switch the communications RAM into the controller before requesting any service which uses it (e.g., monitor, force, timer/counter preset/examine modes). Only after the switch has been made does the loader set the mode code and generate an interrupt.

The controller needs to acknowledge receipt of the loader's interrupt signal as well as report when it has finished servicing the communications or program RAM. These two functions are combined into one "memory busy" signal. After the loader has switched the program RAM and communications RAM as required by the particular mode to be entered (control register A bits 4 and 5) and has set the proper mode code (data register B), it generates an interrupt on its CB2 line which is hard-wired to the controller's CB1 line. Having generated the interrupt, the loader goes into a timed wait loop and awaits an interrupt acknowledge. Meanwhile, the controller senses the interrupt on its CB1 line, posts the interrupt, finishes processing the current memory scan, and then begins interrupt processing. If the request for service was only a mode change to or from "run from RAM" mode, the controller makes no specific response other than to honor the request; in all other modes, either the program RAM or the communications RAM, or both, must be shared, and hence synchronized.

When interrupt processing begins, the controller generates a combination "interrupt acknowledge" and "memory busy" signal by setting its CB2 line high. The controller's CB2 line is hard-wired to loader PIA data register A, bit 0. A low-to-high transition acknowledges that the interrupt was received and is being serviced (the memory is busy). When the controller has finished and no longer needs the memory, it sets CB2 low. The loader, watching CB2 through data register A, bit 0, sees the high-to-low transition and recognizes that interrupt service is complete; the loader may now switch the memory back into the loader for further processing.

The timing diagram in Figure 20 summarizes the synchronization signals.

6.5 PIA Pin Assignments

Below is a summary of the loader and controller PIA bit assignments.

LOADER PIA #1 (Communications PIA)

41FC - data register A

bit 0 - interrupt acknowledge/memory busy signal from controller

bit 1 - spare

bit 2 - spare

bit 3 - nibble select

bit 4 - communications RAM switch (0 = loader, 1 = controller)

bit 5 - program RAM switch (0 = loader, 1 = controller)

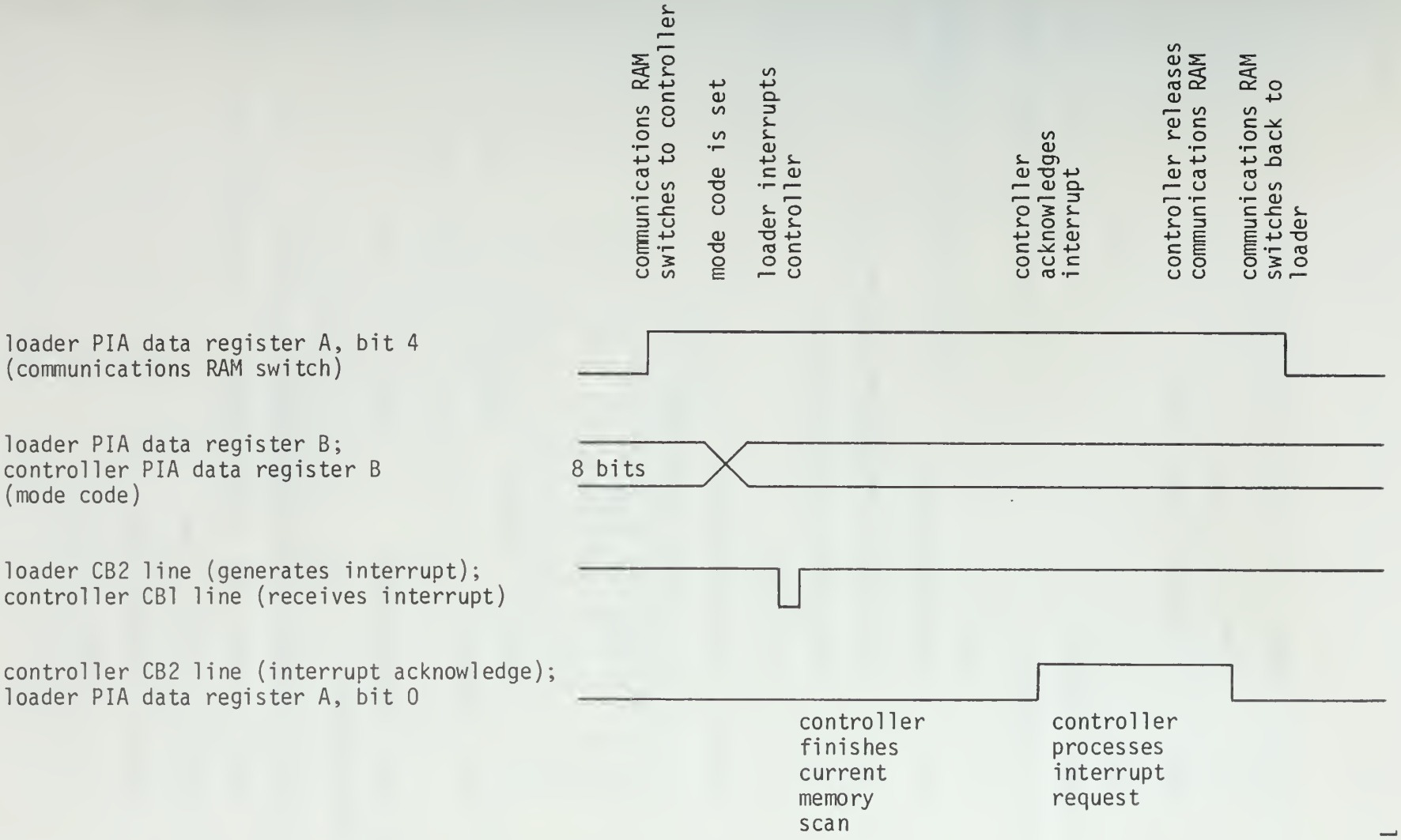


Figure 20. Timing Diagram for Interrupt Service

bit 6 - spare

bit 7 - spare

CA1 - CRT video vertical retrace

CA2 - spare

41FD - control register A

41FE - data register B

bit 0 - unused

bit 1 - copy program mode

bit 2 - timer/counter preset/count examine mode

bit 3 - timer/counter preset set mode

bit 4 - force mode

bit 5 - run from RAM mode

bit 6 - monitor one location mode

bit 7 - monitor all mode

CB1 - spare

CB2 - generate interrupt to controller

41FF - control register B

LOADER PIA #2 (internal use)

4018 - data register A

bit 0 - address A8 of PROM socket

bit 1 - address A9 of PROM socket

bit 2 - program pulse

bit 3 - RAM chip enable

bit 4 - PROM chip enable

bit 5 - read/write protect switch

bit 6 - read/write clear switch

bit 7 - beeper

CA1 - spare

CA2 - spare

4019 - control register A

CONTROLLER PIA (Communications)

4014 - data register A

unused

4015 - control register A

4016 - data register B

bit 0 - unused

bit 1 - copy program mode

bit 2 - timer/counter preset/count examine mode

bit 3 - timer/counter preset set mode

bit 4 - force mode

bit 5 - run from RAM mode

bit 6 - monitor one location mode

bit 7 - monitor all mode

CB1 - receives interrupt from loader

CB2 - interrupt acknowledge/memory busy signal to loader

4017 - control register B

6.6 Mode Descriptions

6.6.1 COPY PROGRAM Mode

When the loader wishes to monitor a controller executing a PROM program, the loader must first make a copy of the program in its program RAM so that it may be decompiled. The COPY PROGRAM mode causes the controller to copy its program in PROM (2000 - 23FF) into the program RAM area (3000 - 33FF). The controller assumes that the program RAM has been assigned to the controller prior to this request.

In COPY PROGRAM mode the "memory busy" signal generated by the controller on its CB2 line reflects the busy status of the program RAM. If this mode is used in combination with another mode requiring synchronization of the communications RAM as well, the "memory busy" signal will be the logical OR of the busy status of the program RAM and communications RAM.

Sequence:

1. Loader switches program RAM into controller.
2. Loader sets COPY PROGRAM mode.
3. Loader interrupts controller.
4. Controller recognizes interrupt, saves new mode, continues processing.
5. Loader awaits interrupt acknowledge/memory busy signal. If not received with 0.1 seconds, loader issues an error message.
6. At end of scan controller processes COPY PROGRAM mode.

7. Controller acknowledges interrupt and signals program RAM busy by setting memory busy signal high (CB2).
8. Loader recognizes memory busy and awaits memory free. If not received within 0.1 seconds, loader issues an error message.
9. Controller copies PROM into program RAM.
10. Controller signals memory free (CB2 low).
11. Controller cancels COPY PROGRAM mode.
12. Loader recognizes memory free and switches program RAM back into loader for further processing.

6.6.2 MONITOR ALL Mode

The MONITOR ALL mode causes the controller to provide the loader with a copy of the lower halves of the controller's pages zero and one (controller's system data areas). From this data base the loader can determine the status of every input, output, control relay, timer, counter, etc., in the controller.

Sequence:

1. Loader switches communications RAM to controller.
2. Loader set MONITOR ALL mode.
3. Loader interrupts controller.
4. Controller recognizes interrupt, saves new mode, continues processing.
5. Loader awaits interrupt acknowledge/memory busy signal. If not received within 0.1 seconds, loader issues an error message.

6. At end of scan controller processes MONITOR ALL request.
7. Controller acknowledges interrupt and signals communications RAM busy.
8. Loader recognizes memory busy and awaits memory free. If not received within 0.1 seconds, loader issues an error message.
9. Controller copies locations 00 - 7F of pages 0 and 1 into monitor area.
10. Controller signals communications RAM free.
11. Controller cancels MONITOR ALL request.
12. Loader recognizes memory free and switches communications RAM back into loader for further processing.

6.6.3 MONITOR ONE ADDRESS Mode

The MONITOR ONE ADDRESS mode permits the loader to specify one 16-bit address and have the controller return the contents of the specified location. While originally intended for use with an auxiliary data entry panel, this is a very versatile facility which, in effect, gives the loader the capability of examining any location in the controller.

Sequence:

1. Loader switches communications RAM into controller.
2. Loader sets MONITOR ONE ADDRESS mode.
3. Loader interrupts controller.
4. Controller recognizes interrupt, saves new mode, continues processing.

5. Loader awaits interrupt acknowledge/memory busy signal. If not received within 0.1 seconds, loader issues error message.
6. At end of scan controller processes MONITOR ONE ADDRESS request.
7. Controller acknowledges interrupt and shows communications RAM busy.
8. Loader recognizes memory busy and awaits memory free. If not received within 0.1 seconds, loader issues error message.
9. Controller reads address to be monitored from locations 2FFD (low) and 2FFE (high).
10. Controller returns content (8 bits) of indicated location at address 2FFF.
11. Controller signals memory free.
12. Controller cancels MONITOR ONE ADDRESS mode.
13. Loader recognizes memory free and switches communications RAM back into loader for interrogation.

6.6.4 RUN FROM RAM Mode

During debugging, the user program may be stored in the loader's program RAM. This mode directs the controller to execute the program in RAM rather than the normal PROM program.

The current status of the RUN FROM RAM bit must be correct on every interrupt since both the 0 and 1 states are significant.

Sequence:

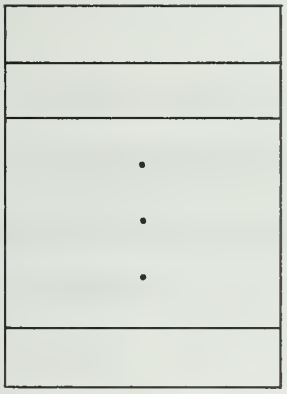
1. Loader switches program RAM into controller.
2. To initiate RUN FROM RAM, loader sets data register B, bit 5 = 1. To terminate RUN FROM RAM, loader sets data register B, bit 5 = 0.
3. Loader interrupts controller.
4. Controller recognizes interrupt, saves new mode, continues processing.
5. Loader waits 0.1 seconds.
6. At the end of every scan during which an interrupt occurred, the controller uses data register B, bit 5, to determine whether the next memory scan will use the program in PROM (bit 5 = 0) or RAM (bit 5 = 1).

6.6.5 FORCE Mode

FORCE mode operates as an override--selected inputs may be forced on or off, while others retain their real world values. The loader prepares a FORCE VECTOR, a group of 64 contiguous bytes whose most significant bit (B7) specifies whether the corresponding input is being forced; if so, bit B6 specifies whether the input is forced on or forced off. After every I/O snapshot, the controller determines if it is operating in force mode; if so, the FORCE VECTOR is used to override the real inputs. Each interrupt from the loader with the FORCE mode bit set causes the controller to copy a new FORCE VECTOR from the communications RAM into the controller's system variable area.

FORCE VECTOR

Address



2F01	FORCE specification for input #1
2F02	FORCE specification for input #2
. . .	. . .
2F40	FORCE specification for input #64

key: B7 { = 0 input not forced (normal)
= 1 input forced

B6 { = 0 input forced off
= 1 input forced on

The current status of the FORCE mode must be correctly set for every interrupt, since both the 0 and 1 states are significant.

Sequence:

1. Loader determines whether to initiate/continue or terminate FORCE mode.

TERMINATE FORCE MODE

2. Loader sets data register B, bit 4 = 0.
3. Loader interrupts controller.
4. Controller recognizes interrupt, saves new mode, continues processing.
5. Loader waits 0.1 seconds.

6. Loader cancels FORCE mode.

7. At end of scan, controller cancels FORCE mode.

INITIATE/CONTINUE FORCE MODE

2. Loader switches communications RAM into loader.

3. Loader prepares FORCE VECTOR in force area (2F01 - 2F40).

4. Loader switches communications RAM into controller.

5. Loader sets FORCE mode (data register B, bit 4 = 1).

6. Loader interrupts controller.

7. Controller recognizes interrupt, saves new mode, continues processing.

8. Loader awaits interrupt acknowledge/memory busy signal. If not received within 0.1 seconds, loader issues error message.

9. At end of scan controller enters/continues FORCE mode.

10. Controller acknowledges interrupt and shows communications RAM busy.

11. Loader recognizes memory busy and awaits memory free. If not received within 0.1 seconds, loader issues error message.

12. Controller copies FORCE VECTOR from force area into controller's system variable area.

13. Controller signals memory free.

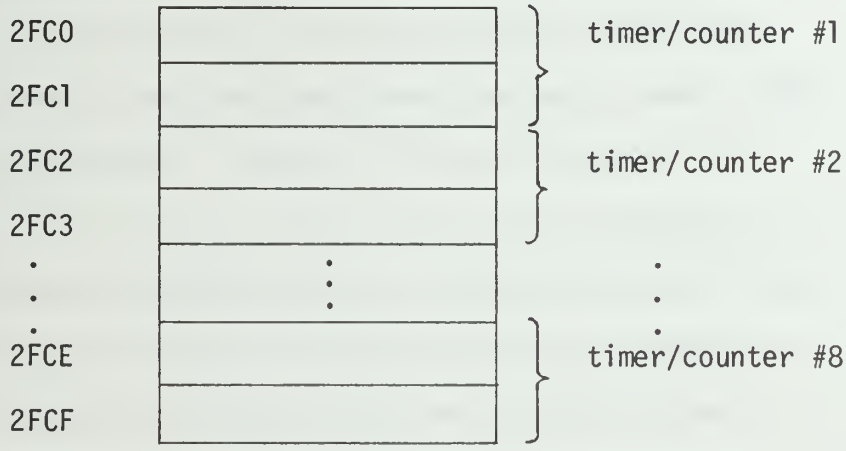
14. Loader recognizes memory free and switches communications RAM back into loader for further processing.

6.6.6 INSERT TIMER/COUNTER PRESET Mode

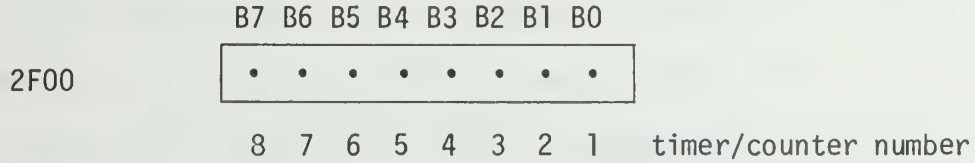
This mode allows the loader to dynamically alter any or all of the 8 timer/counter presets in the controller.

Sequence:

1. Loader switches communications RAM into loader and puts new preset values into their proper locations.



2. Loader sets timer/counter flags to indicate which presets are to be altered.



Only those presets which have a corresponding '1' bit in the flag byte will be inserted into the controller.

3. Loader switches communications RAM into controller.
4. Loader sets INSERT TIMER/COUNTER PRESET mode.
5. Loader interrupts controller.

6. Controller recognizes interrupt, saves new mode, continues processing.
7. Loader awaits interrupt acknowledge/memory busy signal. If not received within 0.1 seconds, loader issues error message.
8. At end of scan controller processes request.
9. Controller acknowledges interrupt and shows communications RAM busy.
10. Loader recognizes memory busy and awaits memory free. If not received within 0.1 seconds, loader issues error message.
11. Controller examines flag byte (2FF0). For each flag bit which is set, the controller transfers the corresponding preset from the preset area (2FC0 - 2FCF) into the controller's system area.
12. Controller signals memory free.
13. Controller cancels INSERT TIMER/COUNTER PRESET mode.
14. Loader recognizes memory free and switches communications RAM back into loader for further processing.

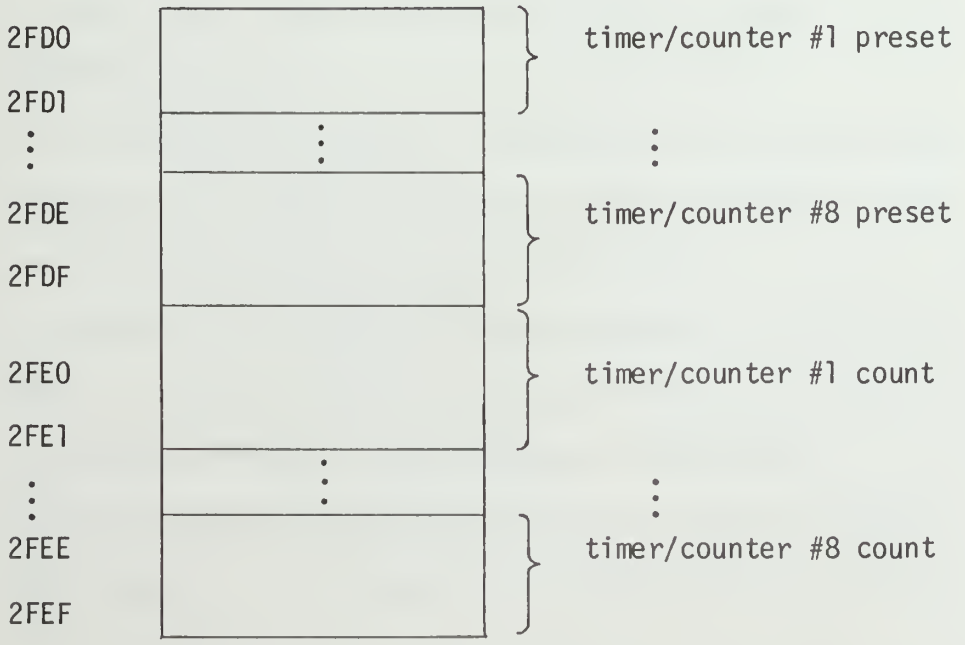
6.6.7 EXAMINE TIMER/COUNTER PRESETS AND COUNTS Mode

This mode allows the loader to monitor the preset and/or current count of any timer or counter.

Sequence:

1. Loader switches communications RAM into controller.

2. Loader sets EXAMINE mode.
3. Loader interrupts controller.
4. Controller recognizes interrupt, saves new mode, continues processing.
5. Loader awaits interrupt acknowledge/memory busy signal. If not received within 0.1 seconds, loader issues error message.
6. At end of scan controller processes EXAMINE mode.
7. Controller acknowledges interrupt and signals communications RAM busy.
8. Loader recognizes memory busy and awaits memory free. If not received within 0.1 seconds, loader issues error message.
9. Controller copies all 8 timer/counter presets and all 8 timer/counter counts from its system area into the communications RAM.



10. Controller signals memory free.
11. Controller cancels EXAMINE mode.
12. Loader recognizes memory free and switches communications RAM back into loader for further processing.

7. UTILITY OF MICROPROCESSORS FOR INDUSTRIAL CONTROL

7.1 Utility

An outstanding question in industrial process control is whether or not future industrial process controllers will abandon discrete-component hardware for microprocessor-based, software-driven controllers. The IEEE sponsored an Industrial Electronics and Control Instrumentation Conference [39] in March, 1976, to examine exactly this question. The participants in the "New Developments in Programmable Controllers" session were primarily electrical engineers, control engineers, and officers from such established industrial controller manufacturers as Modicon, Eagle Signal, Datametrics, and Controlex. Several pertinent points relative to the use of microprocessors in industrial controllers were discussed, including:

1. Programming cost and the complexity of microprocessors increases overall system costs above that of competing dedicated controllers;
2. The programming language of controllers (i.e., Boolean equations or relay ladder diagrams) is better adapted to that of the user, usually the plant electrician;
3. Controllers, in contrast to microcomputers, are generally better designed to withstand hostile physical working environments and harsh radio frequency interference;
4. Microprocessors allow standardization of system hardware and changes may be effected by altering only the program;

5. Because of their slow speed, MOS microprocessors are limited in the number of I/O lines they can handle in a programmable controller configuration.

The work presented here takes issue with three of the four objections and is in agreement with the one reported advantage.

Programming Cost

The programming costs occur in two distinct areas: cost to the manufacturer who must supply the system software (compiler, assembler, operating system, graphic translator, I/O package, etc.) necessary to make the controller function, and cost to the end user who must pay in salaries for whatever level of programming expertise is necessary to utilize the control system. In the former case one cannot argue against the fact that the development costs for a complex software system exceeds the nonexistent software cost of an all-hardware system. As we have seen, however, the evolution of programmable controllers and many of the major advances in controller technology have been due to the availability of sophisticated system software which simply outperforms its hardware equivalent (e.g., software translation of graphically input RLDs vs. hardware translation of Boolean instructions input via pushbuttons and thumbwheel switches).

The latter consideration (continuing cost to the user) illustrates the advantages of the software investment. If system software amounts to no more than the assembly language and assembler provided by

the microprocessor manufacturer, then the development, debugging, and maintenance of a control system written in assembly language by each user will represent a large and continuing investment in programming time and talent. We have chosen to invest in one complex software package which, in turn, makes each user's programming task simpler and less expensive.

Thus we believe that the use of microprocessors will ultimately decrease the cost of programming industrial controllers by supporting sophisticated system software which simplifies user programming.

Programming Language

For those control problems which can be described by a relay ladder diagram, the graphically-input RLD is a clear favorite over either Boolean algebra equations translated into a pseudo-Boolean assembly language or a specific assembly language for a specific microprocessor. The graphic programming capabilities of the D-1001 exceed those of any other controller in the same price range. The great advantage of RLD programming is that it is usually the most appropriate input medium for the majority of prospective users (plant electricians).

We refute the idea that microprocessors can be successfully user-programmed only in microprocessor assembly language; we have demonstrated conclusively that microprocessors will support graphic (RLD) programming.

Noise Immunity

As discussed in Chapter 1, achieving adequate noise immunity is

basically a hardware problem and as such is outside the scope of this thesis. However, the hardware isolation techniques used, coupled with the small size of the total microprocessor system which makes external shielding cost-effective, appears to have solved the noise problem in many typical applications. As an additional safeguard, the software is capable of detecting and recovering from some noise-induced errors.

Software Flexibility

The system software has, as expected, gone through a number of iterations before reaching steady state (the controller alone has seen some 30 revisions of its operating system). While some new versions were merely error corrections, most represented attempts to change the basic character of the programming and control system by altering only system software in PROM.

Consider the possibilities. As with any hardware, the design must be finalized at some point so that parts can be bought and the unit can go into production. Assume that an initial production run of controllers is delivered and users report dissatisfaction with, say, the permissible range of the system timers, and a decision is made to extend their range from 999.9 seconds to 9999.9 seconds. What will this require? Had the timers been implemented in hardware, such a change would be catastrophic; the production of a new controller with an extended timing range would require a new hardware design, new artwork, new components, and changes to the manufacturing procedure.

In a software-driven system, such a change could be instituted by simply reprogramming the timer module of the operating system. This particular change could be accomplished in approximately one hour of programming time and another hour of simulation on the complete, interactive, controller system simulator developed especially for this purpose. Such software tools as editors, assemblers, emulators, and debug routines make the development of, and changes to, operating system modules a relatively simple task. Whenever a new version of the system software is generated, we produce paper tapes which are used directly for programming the operating system PROMs. Significant changes to the controller's operating system are routinely implemented in half a day.

Again, the flexibility of being able to change the entire character of the programming system by merely changing the system software has proved to be infinitely superior to changing the hardware every week.

Speed

A valid concern is the speed at which microprocessor-based controllers can operate. MOS microprocessors are limited to the number of I/O lines they can handle in a programmable controller configuration and still maintain an adequate system response time. As discussed in Chapter 3, the benchmark program, translated into directly executable microprocessor code, would require approximately 4 ms per program iteration as compared to the 20 to 25 ms required by interpretive execution (although "average" execution-time optimization could be reasonably expected to reduce the solution time to 14 to 20 ms). Still, the 4 ms figure represents a reasonable lower bound for this system's minimum response time.

This imposes a limitation on both the number of inputs and outputs which a controller can reasonably handle and on the ultimate length of a user program.

In summary, we may make the following generalizations concerning the future of microprocessor-based industrial controllers.

1. A principal advantage of conventional programmable controllers is that programming costs are low from the viewpoint of both system designer and end user. Although microprocessors do increase programming costs for the system designers, the tremendous flexibility of a software-driven system is worth the expense.

2. The same hardware will handle many different types of control situations when it can be customized via software changes, rather than requiring additional, custom-designed hardware.

3. The software packages can be standardized and made available off the shelf. Microprocessor-based programming systems are fully capable of supporting graphically-input relay ladder diagrams, or a pseudo-Boolean assembly language, or assembly language for a given microprocessor, or directly executable microprocessor code. Traditional tradeoffs between execution time and memory space prevail.

4. If one temporarily disregards the complexity of programming, and considers only the implementation of the control logic for discrete on-off signals, it could be argued that the circuit complexity and programming expense of a microprocessor exceed that of the equivalent function implemented in, say, discrete CMOS logic [25,39]. However, when one attempts to depart from rudimentary Boolean control and to expand to timing,

counting, arithmetic, or analog sensing, the microprocessor approach is more appropriate. When one considers programming flexibility from the user's point of view, the advantage lies clearly with the microprocessor.

5. Finally, regarding the ultimate speed (and, hence, capacity) of microprocessor-based controllers, it seems clear that continued technological progress in the areas of bit-sliced, microprogrammable microprocessors may yet produce a CPU with suitable speed and noise tolerance properties.

7.2 Future Areas of Research

No attempt is made to claim that either the choice of the particular microprocessor or the particular interpretive code generated by the graphic translator from the input RLD are in any sense the "best possible." Both, however, represent reasonable compromises among many choices and tradeoffs, both hardware and software. As hardware technology changes and as system software capabilities advance, the questions of what type microprocessor to use, what type(s) of user programming to support, and what kind of internal code to generate must again be evaluated.

The choices made in designing the D-1001 have proven reasonable and workable. It is anticipated that, at least for the immediate future, any newer version (D-1002?) would differ from the D-1001 primarily with regard to the system architecture, rather than in either the user programming concepts (e.g., interactive programming, graphic input) or the basic system software characteristics (e.g., the interpretable, column-oriented internal code).

The emphasis will probably be on the replacement of the many discrete components of the hardware (memories, latches, line drivers, etc.) with newer, more versatile, and, not surprisingly, software-driven hardware, such as the MOS Technology 6530 Peripheral Interface/Memory Device which combines a 1K ROM, 64 bytes of RAM, a PIA (including two control registers and two software-controlled bi-directional data ports, and a programmable timer). A new system architecture using 6530s could conceivably reduce the chip count for the 32 I/O system from its current twenty-seven to approximately eleven.

7.3 Conclusion

The goals of the D-1001 controller and program loader are to provide a nontechnical user with a fast, efficient, simple to use, cost-effective system for implementing control of sequential processes. The choices of microprocessor-based hardware, interactive graphic programming in a relay symbol language, and development of sophisticated system software to implement system functions such as programming, editing, monitoring, and debugging, are consistent with current hardware and software technology and with the surveyed needs of the intended users of such a system. Initial reports concerning the first production units in the field indicate that we have successfully achieved these goals.

REFERENCES

[1] "VIP Programmable Controller Instruction Manual," part number 79608, Struthers-Dunn, Inc., Bettendorf, Iowa 52722, 1973.

[2] "VIP Operating and Programming Manual," manual number 2500, Struthers-Dunn, Inc., Bettendorf, Iowa 52722, 1972.

[3] "Electrical Standards for Mass Production Equipment," JIC Electrical Standard #EMP-1-1967, Joint Industrial Council, Washington, DC 20007, 1967.

[4] "Graphic Symbols for Electrical and Electronics Diagrams," Standard Y32.2, United States of America Standards Institute, New York, NY 10016.

[5] "Standard IC-1, Industrial Control," National Electrical Manufacturers Association, New York, NY 10017.

[6] "pc^2 Language Primer," Fisher Controls Company, 1970.

[7] Duncan, J. B., "Anatomy of pc^2: Process Control Language," Control Engineering, April 1974, pp. 42-44.

[8] Pike, H. E., "Process Control Software," Proceedings of the IEEE, Vol. 58, No. 1, January 1970, pp. 87-97.

[9] Schoeffler, J. D., and Temple, R. H., "A Real-Time Language for Industrial Process Control," Proceedings of the IEEE, Vol. 58, No. 1, January 1970, pp. 98-111.

[10] Saba, Richard T., "Pushbutton Wiring Replaces Relays," Sixth Annual Meeting of the IEEE Industry and Applications Group, Paper D-7095, 1971.

[11] Symmes, David T., "Programmable Controllers and Computers Complement Each Other for More Effective Industrial Control," Computer Design, September 1973, pp. 54-60.

[12] "Modicon 084 Controller," Technical Reference 3-1, Modicon Corporation, Andover, MA 01810, 1971.

[13] "Introduction to Models 184/384," Modicon Corporation, Andover, MA 01810, 1976.

[14] "The Modicon 284 Programmable Controller," Modicon Corporation, Andover, MA 01810, 1973.

[15] "Industrial 14 Systems Manual," Manual number DEC-14-HSMAA-A-D, Digital Equipment Corporation, Maynard, MA 01754, 1972.

[16] "CRT-14 Control Relay Translator," Industrial Control Products Division, Digital Equipment Corporation, Maynard, MA 01754, 1971.

[17] "VT-14 User's Manual," Manual number DEC-14-GFTMA-A-D, Digital Equipment Corporation, Maynard, MA 01754, 1973.

[18] "Controlpac 600," Bulletin 530, Eagle Signal, Davenport, IA 52803, 1973.

[19] "The EPTAK Advanced Programmable Controller," Eagle Signal, Davenport, IA 52803, 1975.

[20] "PMC Programmable Matrix Controller," Bulletin 1750, Allen Bradley Company, Milwaukee, WI 53204, 1971.

[21] "PDQ-II Standard Programmable Controller," Bulletin 1760, Allen-Bradley Company, Milwaukee, WI 53204, 1973.

[22] "IPC-4000 Industrial Programmable Controller," Industrial Solid State Controls, York, PA 17405, 1973.

[23] "E.A.R.O.M. Programmable Controllers," FX Systems Corporation, Kingston, NY 12401, 1975.

[24] "Automate 33," Bulletin DO2580-2, Reliance Electric Company, Cleveland, OH 44117, 1973.

[25] "S-D 77 Programmable Controller," Struthers-Dunn, Inc., Bettendorf, IA 52722, 1974.

[26] Henry, Donald E., Chief Engineer, Struthers-Dunn, Inc., Bettendorf, IA 52722, personal communication, 1976.

[27] Weaver, Alfred C., VIPTRAN - A Programming Language and its Compiler for Boolean Systems of Process Control Equations, M.S. thesis, Department of Computer Science Report No. UIUCDCS-R-73-603, University of Illinois, Urbana, IL 61801, November 1973.

[28] Weaver, Alfred C., VIPTRAN2 - An Improved Programming Language and its Compiler for Process Control Equations, Department of Computer Science Report No. UIUCDCS-R-74-643, University of Illinois, Urbana, IL 61801, May 1974.

[29] Aho, Alfred V., and Ullman, Jeffrey D., The Theory of Parsing, Translation, and Compiling, Vol. 1, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1972.

[30] Hopgood, F.R.A., Compiling Techniques, American Elsevier, New York, NY 10017, 1969.

[31] Gries, David, Compiler Construction for Digital Computers, John Wiley & Sons, Inc., New York, NY, 1971.

[32] "PDP-8 Small Computer Handbook," Digital Equipment Corporation, Maynard, MA 01754, 1973.

[33] "M6800 Microprocessor Programming Manual," Motorola Semiconductor Products, Inc., 1975.

[34] "Intel Data Catalog," Intel Corporation, Santa Clara, CA 95051, 1975.

[35] "Programming Manual," MOS Technology, Inc., Norristown, PA 19401, 1976.

[36] "Hardware Manual," MOS Technology, Inc., Norristown, PA 19401, 1976.

[37] "IMP-16 Applications Manual," National Semiconductor Corporation, Santa Clara, CA 95051, 1973.

[38] "LSI-11 Microcomputer," Digital Equipment Corporation, Maynard, Ma 01754, 1976.

[39] "Limited Role Seen for Microprocessors in Programmable Controllers," Electronic Design, March 1, 1976, pp. 15.

VITA

The author, Alfred Charles Weaver, was born July 18, 1949, in Johnson City, Tennessee. He was graduated from the University of Tennessee at Knoxville with a Bachelor of Science (summa cum laude) in Engineering Science in March, 1971. He subsequently received the degree of Master of Science in Computer Science from the University of Illinois at Urbana-Champaign in October, 1973. Since September, 1971, he has been employed as a graduate teaching assistant in the Department of Computer Science at the University of Illinois and in 1975 was awarded an IBM Graduate Fellowship in Computer Science. He is a member of Phi Eta Sigma, Tau Beta Pi, Phi Kappa Phi, and Sigma Xi honorary fraternities, is a member of the Association for Computing Machinery, and is a three-year past president of the University of Illinois Student Chapter of the ACM. Currently, he is a Visiting Assistant Professor in the Department of Computer Science of the University of Illinois at Urbana-Champaign.

BLIOGRAPHIC DATA EET	1. Report No. UIUCDCS-R-77-865	2.	3. Recipient's Accession No.
Title and Subtitle A GRAPHICALLY-PROGRAMMED, MICROPROCESSOR-BASED INDUSTRIAL CONTROLLER			5. Report Date May 1977 6.
Author(s) Alfred Charles Weaver			8. Performing Organization Rept. No.
Performing Organization Name and Address DEPARTMENT OF COMPUTER SCIENCE UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN URBANA, ILLINOIS 61801			10. Project/Task/Work Unit No. 11. Contract/Grant No.
Sponsoring Organization Name and Address DEPARTMENT OF COMPUTER SCIENCE UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN URBANA, ILLINOIS 61801			13. Type of Report & Period Covered Ph.D. Thesis 14.

Supplementary Notes

Abstracts

This report discusses the applicability of microprocessors to an industrial vironment, traces the development of process control software, and suggests a new, aphic programming language based on familiar relay symbols as the system's input. design is presented for two stand-alone, microprocessor-based machines--the controller self, which interprets a user program and manages system I/O, and an auxiliary pro- am loeader which supervises interactive graphic programming using a special keyboard d CRT. When connected, the two units communicate to provide the user with the pability of monitoring and changing a running system.

Key Words and Document Analysis. 17a. Descriptors

microprocessor
graphic programming
relay ladder diagram
industrial process control

Identifiers/Open-Ended Terms

COSATI Field/Group

Availability Statement	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 169
	20. Security Class (This Page UNCLASSIFIED	22. Price

USCOMM-DC 40329-P71